FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

Contents

CHAPTER

4

KNOWLEDGE REPRESENTATION ISSUES

In general we are least aware of what our minds do best.

—**Marvin Minsky**
(1927-), American cognitive scientist

In Chapter 1, we discussed the role that knowledge plays in AI systems. In succeeding chapters up until now, though, we have paid little attention to knowledge and its importance as we instead focused on basic frameworks for building search-based problem-solving programs. These methods are sufficiently general that we have been able to discuss them without reference to how the knowledge they need is to be represented. For example, in discussing the best-first search algorithm, we hid all the references to domain-specific knowledge in the generation of successors and the computation of the *h'* function. Although these methods are useful and form the skeleton of many of the methods we are about to discuss, their problem-solving power is limited precisely because of their generality. As we look in more detail at ways of representing knowledge, it becomes clear that particular knowledge representation models allow for more specific, more powerful problem-solving mechanisms that operate on them. In this part of the book, we return to the topic of knowledge and examine specific techniques that can be used for representing and manipulating knowledge within programs.

4.1 REPRESENTATIONS AND MAPPINGS

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. But before we can talk about them individually, we must consider the following point that pertains to all discussions of representation, namely that we are dealing with two different kinds of entities:

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

- The *knowledge level,* at which facts (including each agent's behaviors and current goals) are described.

- The *symbol level,* at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

See Newell [1982] for a detailed exposition of this view in the context of agents and their goals and behaviors. In the rest of our discussion here, we will follow a model more like the one shown in Fig. 4.1. Rather than thinking of one level on top of another, we will focus on facts, on representations, and on the two-way mappings that must exist between them. We will call these links *representation mappings.* The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

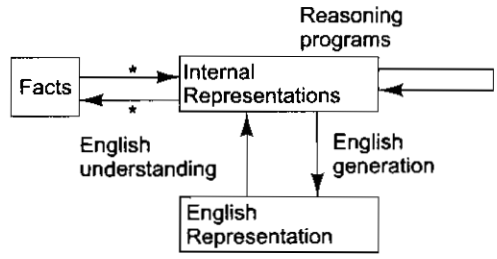


Fig. 4.1 *Mappings between Facts and Representations*

One representation of facts is so common that it deserves special mention: natural language (particularly English) sentences. Regardless of the representation for facts that we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. In this case, we must also have mapping functions from English sentences to the representation we are actually going to use and from it back to sentences. Figure 4.1 shows how these three kinds of objects relate to each other.

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

The fact represented by that English sentence can also be represented in logic as:

$dog(Spot)$

Suppose that we also have a logical representation of the fact that all dogs have tails:

$\forall x : dog(x) \rightarrow hastail(x)$

Then, using the deductive mechanisms of logic, we may generate the new representation object:

$hastail(Spot)$

Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

Or we could make use of this representation of a new fact to cause us to take some appropriate action or to derive representations of additional facts.

It is important to keep in mind that usually the available mapping functions are not one-to-one. In fact, they are often not even functions but rather many-to-many relations. (In other words, each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range.) This is particularly true of the mappings involving English representations of facts. For example, the two sentences "All dogs have tails" and "Every dog has a tail" could both represent the same fact, namely,

that every dog has at least one tail. On the other hand, the former could represent either the fact that every dog has at least one tail or the fact that each dog has several tails. The latter may represent either the fact that every dog has at least one tail or the fact that there is a tail that every dog has. As we will see shortly, when we try to convert English sentences into some other representation, such as logical propositions, we must first decide what facts the sentences represent and then convert those facts into the new representation.

The starred links of Fig. 4.1 are key components of the design of any knowledge-based program. To see why, we need to understand the role that the internal representation of a fact plays in a program. What an AI program does is to manipulate the internal representations of the facts it is given. This manipulation should result in new structures that can also be interpreted as internal representations of facts. More precisely, these structures should be the internal representations of facts that correspond to the answer to the problem described by the starting set of facts.

Sometimes, a good representation makes the operation of a reasoning program not only correct but trivial. A well-known example of this occurs in the context of the mutilated checker board problem, which can be stated as follows:

> **The Mutilated Checker board Problem.** Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?

One way to solve this problem is to try to enumerate, exhaustively, all possible tilings to see if one works. But suppose one wants to be more clever. Figure 4.2 shows three ways in which the mutilated checker board could be represented (to a person). The first representation does not directly suggest the answer to the problem. The second may; the third does, when combined with the single additional fact that each domino must cover exactly one white square and one black square. Even for human problem solvers a representation shift may make an enormous difference in problem-solving effectiveness. Recall that we saw a slightly less dramatic version of this phenomenon with respect to a problem-solving program in Section 1.3.1, where we considered two different ways of representing a tic-tac-toe board, one of which was as a magic square.

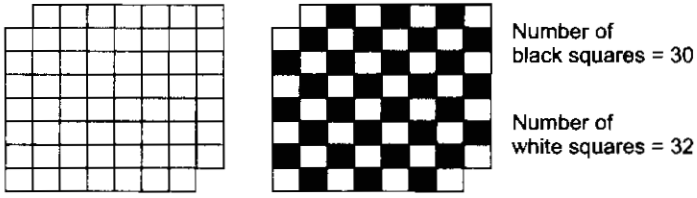


Fig. 4.2 *Three Representations of a Mutilated Checker board*

Figure 4.3 shows an expanded view of the starred part of Fig. 4.1 The dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated. If either the program's operation or one of the representation mappings is not faithful to the problem that is being modeled, then the final facts will probably not be the desired ones. The key role that is played by the nature of the representation mapping is apparent from this figure. If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

It is interesting to note that Fig. 4.3 looks very much like the sort of figure that might appear in a general programming book as a description of the relationship between an abstract data type (such as a set) and a concrete implementation of that type (e.g., as a linked list of elements). There are some differences, though, between this figure and the formulation usually used in programming texts (such as Aho *et al.* [1983]). For example, in data type design it is expected that the mapping that we are calling the backward representation mapping is a function (i.e., every representation corresponds to only one fact) and that it is onto (i.e., there is at least one representation for every fact). Unfortunately, in many AI domains, it may not be possible to come up with such a representation mapping, and we may have to live with one that gives less ideal results. But the main idea of what we are doing is the same as what programmers always do, namely to find concrete implementations of abstract concepts.

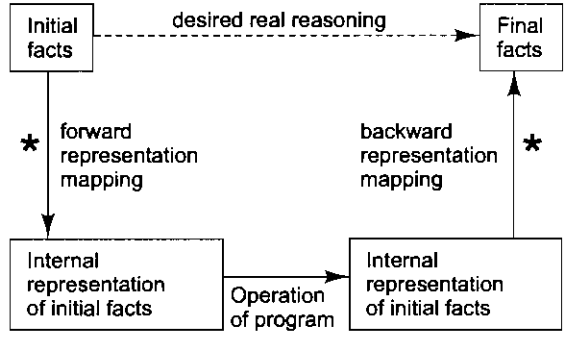


Fig. 4.3 *Representation of Facts*

4.2 APPROACHES TO KNOWLEDGE REPRESENTATION

A good system for the representation of knowledge in a particular domain should possess the following four properties:

Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.

- Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency — the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mecha- nisms in the most promising directions.
- Acquisitional Efficiency — the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

Simple Relational Knowledge

The simplest way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 4.4 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right
player_info('hank aaron', '6-0', 180,right-right).			

Fig. 4.4 *Simple Relational Knowledge and a sample fact in Prolog*

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Fig. 4.4, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer. If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness, say), then this same relation can provide at least some of the information required by those rules.

Providing support for relational knowledge is what database systems are designed to do. Thus we do not need to discuss this kind of knowledge representation structure further here. The practical issues that arise in linking a database system that provides this kind of support to a knowledge representation system that provides some of the other capabilities that we are about to discuss have already been solved in several commercial products.

Inheritable Knowledge

The relational knowledge of Fig. 4.4 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with inference mechanisms that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is *property inheritance*, in which elements of specific classes inherit attributes and values from more general classes in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Figure 4.5 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. Boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line, The structure shown in the figure is a *slot-and-filler structure.* It may also be called a *semantic network* or a collection *of frames.* In the latter case, each individual frame represents the collection of attributes and values associated with a particular node. Figure 4.6 shows the node for baseball player displayed as a frame.

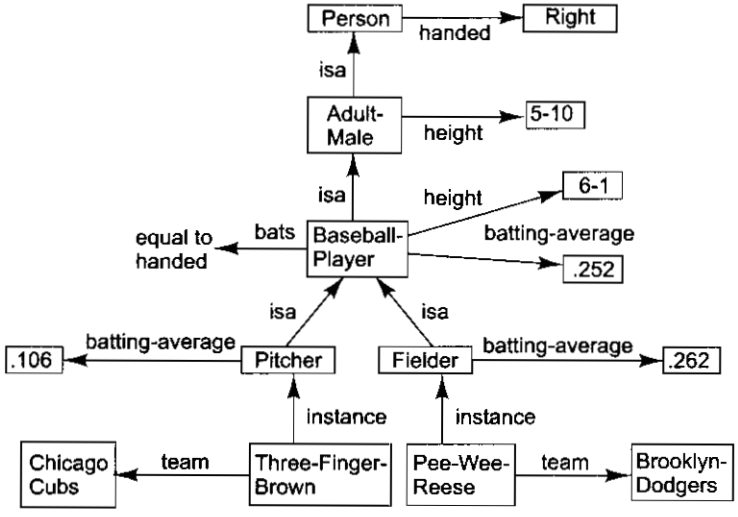


Figure 4.5 *Inheritable Knowledge*

Baseball-Player	
isa:	*Adult-Male*
bats:	(EQUAL *handed*)
height:	6-1
batting-average:	.252

Fig. 4.6 *Viewing a Node as a Frame*

Do not be put off by the confusion in terminology here. There is so much flexibility in the way that this (and the other structures described in this section) can be used to solve particular representation problems that it is difficult to reserve precise words for particular representations. Usually the use of the term *frame system* implies somewhat more structure on the attributes and the inference mechanisms that are available to apply to them than does the term *semantic network.*

In Chapter 9 we discuss structures such as these in substantial detail. But to get an idea of how these structures support inference using the knowledge they contain, we discuss them briefly here. All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute *isa,* which is being used to show class inclusion, and the attribute *instance,* which is heing used to show class membership. These two specific (and generally useful) attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have heen explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows:

Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A.* If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attrihute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A.* If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or *isa* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- *team(Pee-Wee-Reese)* = *Brooklyn-Dodgers.* This attribute had a value stored explicitly in the knowledge base.
- *batting-average(Three-Finger Brown)* = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.
- *height(Pee-Wee-Reese)* = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.

- *bats(Three-Finger-Brown) = Right.* To get a value for the attribute *bats* required going up the *isa* hierarchy to the class *Baseball-Player.* But what we found there was not a value but a rule for computing a value. This rule required another value (that for *handed)* as input. So the entire process must be begun again recursively to find a value for *handed.* This time, it is necessary to go all the way up to *Person* to discover that the default value for handedness for people is *Right.* Now the rule for *bats* can be applied, producing the result *Right.* In this case, that turns out to be wrong, since Brown is a switch hitter (i.e., he can hit both left-and right-handed).

Inferential Knowledge

Property inheritance is a powerful form of inference, but it is not the only useful form. Sometimes all the power of traditional logic (and sometimes even more than that) is necessary to describe the inferences that are needed. Figure 4.7 shows two examples of the use of first-order predicate logic to represent additional knowledge about baseball.

$$\forall x : Ball(x) \wedge Fly(x) \wedge Fair(x) \wedge Infield\text{-}Catchable\ (x) \wedge$$
$$Occupied\text{-}Base(First) \wedge Occupied\text{-}Base(Second) \wedge (Outs < 2) \wedge$$
$$\neg[Line\text{-}Drive(x) \vee Attempted\text{-}Bt,(x)]$$
$$\rightarrow Infield\text{-}Fly(x)$$

$$\forall x,y : Batter(x) \wedge batted(x, y) \wedge Infield\text{-}Fly(y) \rightarrow \text{Out}(x)$$

Fig. 4.7 *Inferential Knowledge*

Of course, this knowledge is useless unless there is also an inference procedure *that can exploit* it *(just as* the default knowledge in the previous example would have been useless without our algorithm for moving through the knowledge structure). The required inference procedure now is one that implements the standard logical rules of inference. There are many such procedures, some of which reason forward from given facts to conclusions, others of which reason backward from desired conclusions to given facts. One of the most commonly used of these procedures is *resolution,* which exploits a proof by contradiction strategy. Resolution is described in detail in Chapter 5.

Recall that we hinted at the need for something besides stored primitive values with the *bats* attribute of our previous example. Logic provides a powerful structure in which to describe relationships among values. It is often useful to combine this, or some other powerful description language, witb an *isa* hierarchy. In general, in fact, all of the techniques we are describing here should not be regarded as complete and incompatible ways of representing knowledge. Instead, they should be viewed as building blocks of a complete representational system.

Procedural Knowledge

So far, our examples of baseball knowledge have concentrated on relatively static, declarative facts. But another, equally useful, kind of knowledge is operational, or procedural knowledge, that specifies what to do when. Procedural knowledge can be represented in programs in many ways. The most common way is simply as code (in some programming language such as LISP) for doing something. The machine uses the knowledge when it executes the code to perform a task. Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of inferential adequacy (because it is very difficult to write a program that can reason about another program's behavior) and acquisitional efficiency (because the process of updating and debugging large pieces of code becomes unwieldy).

As an extreme example, compare the representation of the way to compute the value of *bats* shown in Fig. 4.6 to one in LISP shown in Fig. 4.8. Although the LISP one will work given a particular way of storing attributes and values in a list, it does not lend itself to being reasoned about in the same straightforward way

as the representation of Fig. 4.6 does. The LISP representation is slightly more powerful since it makes explicit use of the name of the node whose value for *handed* is to be found. But if this matters, the simpler representation can be augmented to do this as well.

```
Baseball-Player
    isa:            Adult-Male
    bats:           (lambda (x)
                        (prog ()
                            L1
                          (cond ((caddr x) (return (caddr x)))
                                (t (setq x (eval (cadr x)))
                                   (cond (x (go L1))
                                         (t (return nil)))))))
    height:         6-1
    batting-average: .252
```

Fig. 4.8 *Using LISP Code to Define a Value*

Because of this difficulty in reasoning with LISP, attempts have been made to find other ways of representing procedural knowledge so that it can relatively easily be manipulated both by other programs and by people.

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules. Figure 4.9 shows an example of a production rule that represents a piece of operational knowledge typically possessed by a baseball player.

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods discussed in this chapter. But making a clean distinction between declarative and procedural knowledge is difficult. Although at an intuitive level such a distinction makes some sense, at a formal level it disappears, as discussed in Section 6.1. In fact, as you can see, the structure of the declarative knowledge of Fig. 4.7 is not substantially different from that of the operational knowledge of Fig. 4.9. The important difference is in how the knowledge is used by the procedures that manipulate it.

```
If:     ninth inning, and
        score is close, and
        less than 2 outs, and
        first base is vacant, and
        batter is better hitter than next batter,
Then:   walk the batter.
```

Fig. 4.9 *Procedural Knowledge as Rules*

4.3 ISSUES IN KNOWLEDGE REPRESENTATION

Before embarking on a discussion of specific mechanisms that have been used to represent various kinds of real-world knowledge, we need briefly to discuss several issues that cut across all of them:

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?

- At what level should knowledge be represented? Is there a good set of *primitives* into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

We will talk about each of these questions briefly in the next five sections.

4.3.1 Important Attributes

There are two attributes that are of very general significance, and we have already seen their use: *instance* and *isa*. These attributes are important because they support property inheritance. They are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. In slot-and-filler systems, such as those described in Chapters 9 and 10, these attributes are usually represented explicitly in a way much like that shown in Fig. 4.5 and 4.6. In logic-based systems, these relationships may be represented this way or they may be represented implicitly by a set of predicates describing particular classes. See Section 5.2 for some examples of this.

4.3.2 Relationships among Attributes

The attributes that we use to describe objects are themselves entities that we represent. What properties do they have independent of the specific knowledge they encode? There are four such properties that deserve mention here:

- Inverses
- Existence in an *isa* hierarchy
- Techniques for reasoning about values
- Single-valued attributes

Inverses

Entities in the world are related to each other in many different ways. But as soon as we decide to describe those relationships as attributes, we commit to a perspective in which we focus on one object and look for binary relationships between it and others. Attributes are those relationships. So, for example, in Fig. 4.5, we used the attributes *instance*, *isa* and *team*. Each of these was shown in the figure with a directed arrow, originating at the object that was being described and terminating at the object representing the value of the specified attribute. But we could equally well have focused on the object representing the value. If we do that, then there is still a relationship between the two entities, although it is a different one since the original relationship was not symmetric (although some relationships, such as *sibling*, are). In many cases, it is important to represent this other view of relationships. There are two good ways to do this.

The first is to represent both relationships in a single representation that ignores focus. Logical representations are usually interpreted as doing this. For example, the assertion:

team(Pee-Wee-Reese, Brooklyn-Dodgers)

can equally easily be interpreted as a statement about Pee Wee Reese or about the Brooklyn Dodgers. How it is actually used depends on the other assertions that a system contains.

The second approach is to use attributes that focus on a single entity but to use them in pairs, one the inverse of the other. In this approach, we would represent the team information with two attributes:

- one associated with Pee Wee Reese:
  team = *Brooklyn-Dodgers*

- one associated with Brooklyn Dodgers:
  team-members = *Pee-Wee-Reese,...*

This is the approach that is taken in semantic net and frame-hased systems. When it is used, it is usually accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slots by forcing them to be declared and then checking each time a value is added to one attribute that the corresponding value is added to the inverse.

An Isa Hierarchy of Attributes

Just as there are classes of objects and specialized subsets of those classes, there are attributes and specializations of attributes. Consider, for example, the attribute *height*. It is actually a specialization of the more general attribute *physical-size* which is, in turn, a specialization of *physical-attribute*. These generalization-specialization relationships are important for attributes for the same reason that they are important for other concepts—they support inheritance. In the case of attributes, they support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for Reasoning about Values

Sometimes values of attributes are specified explicitly when a knowledge base is created. We saw several examples of that in the baseball example of Fig. 4.5. But often the reasoning system must reason about values it has not been given explicitly. Several kinds of information can play a role in this reasoning, including:

- Information about the type of the value. For example, the value of *height* must be a number measured in a unit of length.
- Constraints on the value, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person's parents.
- Rules for computing the value when it is needed. We showed an example of such a rule in Fig. 4.5 for the *bats* attribute. These rules are called *backward* rules. Such rules have also been called *if-needed rules.*
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called *forward* rules, or sometimes *if-added rules.*

We discuss forward and backward rules again in Chapter 6, in the context of rulebased knowledge representation.

Single-Valued Attributes

A specific but very useful kind of attribute is one that is guaranteed to take a unique value. For example, a baseball player can, at any one time, have only a single height and be a member of only one team. If there is already a value present for one of these attributes and a different value is asserted, then one of two things has happened. Either a change has occurred in the world or there is now a contradiction in the knowledge base that needs to be resolved. Knowledge-representation systems have taken several different approaches to providing support for single-valued attributes, including:

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

4.3.3 Choosing the Granularity of Representation

Regardless of the particular representation formalism we choose, it is necessary to answer the question "At what level of detail should the world be represented?" Another way this question is often phrased is "What should be our primitives?" Should there be a small number of low-level ones or should there be a larger number covering a range of granularities? A brief example illustrates the problem. Suppose we are interested in the following fact:

John spotted Sue.

We could represent this as[1]

spotted(agent(John),
object(Sue))

Such a representation would make it easy to answer questions such as:

Who spotted Sue?

But now suppose we want to know:

Did John see Sue?

The obvious answer is "yes," but given only the one fact we have, we cannot discover that answer. We could, of course, add other facts, such as

$$spotted(x, y) \rightarrow saw(x, y)$$

We could then infer the answer to the question.

An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

saw(agent(John),
object(Sue),
timespan(briefly))

In this representation, we have broken the idea of *spotting* apart into more primitive concepts *of seeing* and *timespan*. Using this representation, the fact that John saw Sue is immediately accessible. But the fact that he spotted her is more difficult to get to.

The major advantage of converting all statements into a representation in terms of a small set of primitives is that the rules that are used to derive inferences from that knowledge need be written only in terms of the primitives rather than in terms of the many ways in which the knowledge may originally have appeared. Thus what is really being argued for is simply some sort of canonical form. Several AI programs, including those described by Schank and Abelson [1977] and Wilks [1972], are based on knowledge bases described in terms of a small number of low-level primitives.

[1] The arguments *agent* and *object* are usually called *cases*. They represent roles involved in the event. This semantic way of analyzing sentences contrasts with the probably more familiar syntactic approach in which sentences have a surface subject, direct object, indirect object, and so forth. We will discuss case grammar [Fillmore, 1968] and its use in natural language understanding in Section 15.3.2. For the moment, you can safely assume that the cases mean what their names suggest.

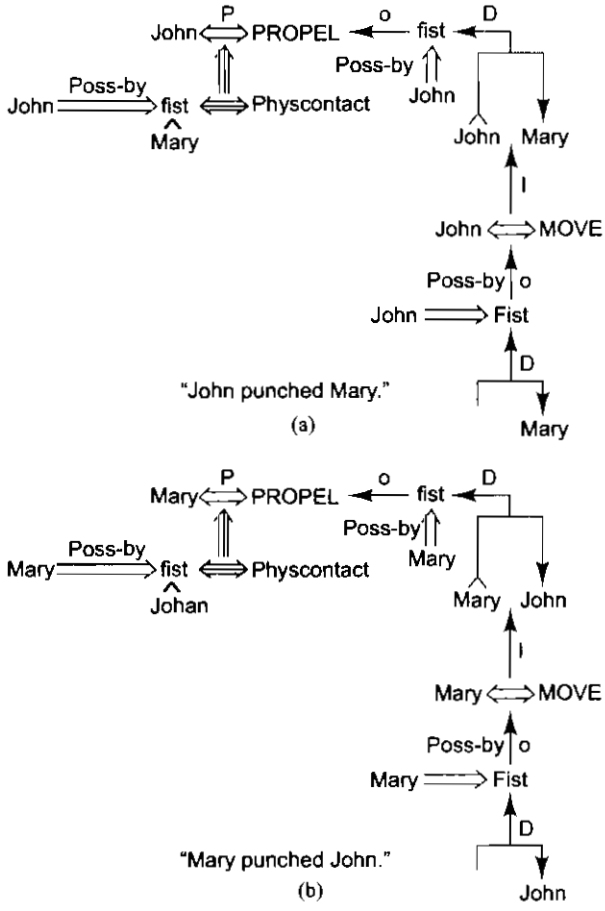


Fig. 4.10 *Redundant Representations*

There are several arguments against the use of low-level primitives. One is that simple high-level facts may require a lot of storage when broken down into primitives. Much of that storage is really wasted since the low-level rendition of a particular high- level concept will appear many times, once for each time the high-level concept is referenced. For example, suppose that actions are being represented as combinations of a small set of primitive actions. Then the fact that John punched Mary might be represented as shown in Fig. 4.10(a). The representation says that there was physical contact between John's fist and Mary. The contact was caused by John propelling his fist toward Mary, and in order to do that John first went to where Mary was.[2] But suppose we also know that Mary punched John. Then we must also store the structure shown in Fig. 4.10(b). If, however, punching were represented simply as punching, then most of the detail of both structures could be omitted from the structures themselves. It could instead be stored just once in a common representation of the concept of punching.

A second but related problem is that if knowledge is initially presented to the system in a relatively high-level form, such as English, then substantial work must be done to reduce the knowledge into primitive form.

[2] The representation shown in this example is called *conceptual dependency* and is discussed in detail in Section 10.1.

Yet, for many purposes, this detailed primitive representation may be unnecessary. Both in understanding language and in interpreting the world that we see, many things appear that later turn out to be irrelevant. For the sake of efficiency, it may be desirable to store these things at a very high level and then to analyze in detail only those inputs that appear to be important.

A third problem with the use of low-level primitives is that in many domains, it is not at all clear what the primitives should be. And even in domains in which there may be an obvious set of primitives, there may not be enough information present in each use of the high-level constructs to enable them to be converted into their primitive components. When this is true, there is no way to avoid representing facts at a variety of granularities.

The classical example of this sort of situation is provided by kinship terminology [Lindsay, 1963]. There exists at least one obvious set of primitives: mother, father, son, daughter, and possibly brother and sister. But now suppose we are told that Mary is Sue's cousin. An attempt to describe the cousin relationship in terms of the primitives could produce any of the following interpretations:

- Mary = *daughter(brother(mother(Sue)))*
- Mary = *daughter(sister(mother(Sue)))*
- Mary = *daughter(brother(father(Sue)))*
- Mary = *daughter(sister(father(Sue)))*

If we do not already know that Mary is female, then of course there are four more possibilities as well. Since in general we may have no way of choosing among these representations, we have no choice but to represent the fact using the nonprimitive relation *cousin*.

The other way to solve this problem is to change our primitives. We could use the set: *parent, child, sibling, male,* and *female*. Then the fact that Mary is Sue's cousin could be represented as

Mary = *child(sibling(parent(Sue)))*

But now the primitives incorporate some generalizations that may or may not be appropriate. The main point to be learned from this example is that even in very simple domains, the correct set of primitives is not obvious.

In less well-structured domains, even more problems arise. For example, given just the fact

John broke the window.

a program would not be able to decide if John's actions consisted of the primitive sequence:

1. Pick up a hard object.
2. Hurl the object through the window.

or the sequence:

1. Pick up a hard object.
2. Hold onto the object while causing it to crash into the window.

or the single action:

1. Cause hand (or foot) to move fast and crash into the window.

or the single action:

1. Shut the window so hard that the glass breaks.

As these examples have shown, the problem of choosing the correct granularity of epresentation for a particular body of knowledge is not easy. Clearly, the lower the level we choose, the less inference required to reason with it in some cases, but the more conference required to create the representation from English and the more room it takes to store, since many inferences will be represented many times. The answer for any particular task domain must come to a large extent from the domain itself—to what use is the knowledge to be put?

One way of looking at the question of whether there exists a good set of low-level primitives is that it is a question of the existence of a unique representation. Does there exist a single, canonical way in which large bodies of knowledge can be represented independently of how they were originally stated? Another, closely related, uniqueness question asks whether individual objects can be represented uniquely and independently of how they are described. This issue is raised in the following quotation from Quine [1961] and discussed in Woods [1975]:

> The phrase *Evening Star* names a certain large physical object of spherical form, which is hurtling through space some scores of millions of miles from here. The phrase *Morning Star* names the same thing, as was probably first established by some observant Babylonian. But the two phrases cannot be regarded as having the same meaning; otherwise that Babylonian could have dispensed with his observations and contented himself with reflecting on the meaning of his words. The meanings, then, being different from one another, must be other than the named object, which is one and the same in both cases.

In order for a program to be able to reason as did the Babylonian, it must be able to handle several distinct representations that turn out to stand for the same object.

We discuss the question of the correct granularity of representation, as well as issues involving redundant storage of information, throughout the next several chapters, particularly in the section on conceptual dependency, since that theory explicitly proposes that a small set of low-level primitives should be used for representing actions.

4.3.4 Representing Sets of Objects

It is important to be able to represent sets of objects for several reasons. One is that there are some properties that are true of sets that are not true of the individual members of a set. As examples, consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world." The only way to represent the facts described in these sentences is to attach assertions to the sets representing people, sheep, and English speakers, since, for example, no single English speaker can be found all over the world. The other reason that it is important to be able to represent sets of objects is that if a property is true of all (or even most) elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every element of the set. We have already looked at ways of doing that, both in logical representations through the use of the universal quantifier and in slot-and-filler structures, where we used nodes to represent sets and inheritance to propagate set-level assertions down to individuals. As we consider ways to represent sets, we will want to consider both of these uses of set-level representations. We will also need to remember that the two uses must be kept distinct. Thus if we assert something like *large(Elephant)*, it must be clear whether we are asserting some property of the set itself (i.e., that the set of elephants is large) or some property that holds for individual elements of the set (i.e., that anything that is an elephant is large).

There are three obvious ways in which sets may be represented. The simplest is just by a name. This is essentially what we did in Section 4.2 when we used the node named *Baseball-Player* in our semantic net and when we used predicates such as *Ball* and *Batter* in our logical representation. This simple representation does make it possible to associate predicates with sets. But it does not, by itself, provide any information about the set it represents. It does not, for example, tell how to determine whether a particular object is a member of the set or not.

There are two ways to state a definition of a set and its elements. The first is to list the members. Such a specification is called an *extensional* definition. The second is to provide a rule that, when a particular object is evaluated, returns true or false depending on whether the object is in the set or not. Such a rule is called an *intensional* definition. For example, an extensional description of the set of our sun's planets on which people live is *{Earth}*. An intensional description is

$$\{x : \textit{sun-planet}(x) \wedge \textit{human-inhabited}(x)\}$$

For simple sets, it may not matter, except possibly with respect to efficiency concerns, which representation is used. But the two kinds of representations can function differently in some cases.

One way in which extensional and intensional representations differ is that they do not necessarily correspond one-to-one with each other. For example, the extensionally defined set *{Earth}* has many intensional definitions in addition to the one we just gave. Others include:

$$\{x : \textit{sun-planet}(x) \ ?\ \textit{nth-farthest-fmm-sun}(x, 3)\}$$
$$\{x : \textit{sun-planet}(x) \ ?\ \textit{nth-biggest}(x, 5)\}$$

Thus, while it is trivial to determine whether two sets are identical if extensional descriptions are used, it may be very difficult to do so using intensional descriptions.

Intensional representations have two important properties that extensional ones lack, however. The first is that they can be used to describe infinite sets and sets not all of whose elements are explicitly known. Thus we can describe intensionally such sets as prime numbers (of which there are infinitely many) or kings of England (even though we do not know who all of them are or even how many of them there have been). The second thing we can do with intensional descriptions is to allow them to depend on parameters that can change, such as time or spatial location. If we do that, then the actual set that is represented by the description will change as a function of the value of those parameters. To see the effect of this, consider the sentence, "The president of the United States used to be a Democrat," uttered when the current president is a Republican. This sentence can mean two things. The first is that the specific person who is now president was once a Democrat. This meaning can be captured straightforwardly with an extensional representation of "the president of the United States." We just specify the individual. But there is a second meaning, namely that there was once someone who was the president and who was a Democrat. To represent the meaning of "the president of the United States" given this interpretation requires an intensional description that depends on time. Thus we might write *president(t)*, where *president* is some function that maps instances of time onto instances of people, namely U.S. presidents.

4.3.5 Finding the Right Structures as Needed

Recall that in Chapter 2, we briefly touched on the problem of matching rules against state descriptions during the problem-solving process. This same issue now rears its head with respect to locating appropriate knowledge structures that have been stored in memory.

For example, suppose we have a script (a description of a class of events in terms of contexts, participants, and subevents) that describes the typical sequence of events in a restaurant.[3] This script would enable us to take a text such as

> John went to Steak and Ale last night. He ordered a large rare steak, paid his bill, and left.

and answer "yes" to the question

[3] We discuss such a script in detail in Chapter 10.

Did John eat dinner last night?

Notice that nowhere in the story was John's eating anything mentioned explicitly. But the fact that when one goes to a restaurant one eats will be contained in the restaurant script. If we know in advance to use the restaurant script, then we can answer the question easily. But in order to be able to reason about a variety of things, a system must have many scripts for everything from going to work to sailing around the world. How will it select the appropriate one each time? For example, nowhere in our story was the word "restaurant" mentioned.

In fact, in order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems.[4]

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situation.
- How to find a better structure if the one chosen initially turns out not to be appropriate.
- What to do if none of the available structures is appropriate.
- When to create and remember a new structure.

There is no good, general purpose method for solving all these problems. Some knowledge-representation techniques solve some of them. In this section we survey some solutions to two of these problems: how to select an initial structure to consider and how to find a better structure if that one turns out not to be a good match.

Selecting an Initial Structure

Selecting candidate knowledge structures to match a particular problem-solving situation is a hard problem; there are several ways in which it can be done. Three important approaches are the following:

- Index the structures directly by the significant English words that can be used to describe them. For example, let each verb have associated with it a structure that describes its meaning. This is the approach taken in conceptual dependency theory, discussed in Chapter 10. Even for selecting simple structures, such as those representing the meanings of individual words, though, this approach may not be adequate, since many words may have several distinct meanings. For example, the word "fly" has a different meaning in each of the following sentences:
  - John flew to New York. (He rode in a plane from one place to another.)
  - John flew a kite. (He held a kite that was up in the air.)
  - John flew down the street. (He moved very rapidly.)
  - John flew into a rage. (An idiom)

  Another problem with this approach is that it is only useful when there is an English description of the problem to be solved.
- Consider each major concept as a pointer to all of the structures (such as scripts) in which it might be involved. This may produce several sets of prospective structures. For example, the concept *Steak* might point to two scripts, one for restaurant and one for supermarket. The concept *Bill* might point to a restaurant and a shopping script. Take the intersection of those sets to get the structure(s), preferably precisely one, that involves all the content words. Given the pointers just described and the story about John's trip to Steak and Ale, the restaurant script would be evoked. One important problem with this method is that if the problem description contains any even slightly extraneous concepts, then the intersection of their associated structures will be empty. This might occur if we had said, for example, "John rode his bicycle to Steak and Ale last night." Another problem is that it may require a great deal of computation to compute all of the possibility sets and then to intersect them. However, if computing such sets and intersecting them could be done in parallel, then the time required to produce an answer

[4] This list is taken from Minsky [1975].

would be reasonable even if the total number of computations is large. For an exploration of this parallel approach to clue intersection, see Fahlman [1979].

- Locate one major clue in the problem description and use it to select an initial structure. As other clues appear, use them to refine the initial selection or to make a completely new one if necessary. For a discussion of this approach, see Charniak [1978]. The major problem with this method is that in some situations there is not an easily identifiable major clue. A second problem is that it is necessary to anticipate which clues are going to be important and which are not. But the relative importance of clues can change dramatically from one situation to another. For example, in many contexts, the color of the objects involved is not important. But if we are told "The light turned red," then the color of the light is the most important feature to consider.

None of these proposals seems to be the complete answer to the problem. It often turns out, unfortunately, that the more complex the knowledge structures are, the harder it is to tell when a particular one is appropriate.

Revising the Choice When Necessary

Once we find a candidate knowledge structure, we must attempt to do a detailed match of it to the problem at hand. Depending on the representation we are using, the details of the matching process will vary. It may require variables to be bound to objects. It may require attributes to have their values compared. In any case, if values that satisfy the required restrictions as imposed by the knowledge structure can be found, they are put into the appropriate places in the structure. If no appropriate values can be found, then a new structure must be selected. The way in which the attempt to instantiate this first structure failed may provide useful cues as to which one to try next. If, on the other hand, appropriate values can be found, then the current structure can be taken to be appropriate for describing the current situation. But, of course, that situation may change. Then information about what happened (for example, we walked around the room we were looking at) may be useful in selecting a new structure to describe the revised situation.

As was suggested above, the process of instantiating a structure in a particular situation often does not proceed smoothly. When the process runs into a snag, though, it is often not necessary to abandon the effort and start over. Rather, there are a variety of things that can be done:

- Select the fragments of the current structure that do correspond to the situation and match them against candidate alternatives. Choose the best match. If the current structure was at all close to being appropriate, much of the work that has been done to build substructures to fit into it will be preserved.
- Make an excuse for the current structure's failure and continue to use it. For example, a proposed chair with only three legs might simply be broken. Or there might be another object in front of it which occludes one leg. Part of the structure should contain information about the features for which it is acceptable to make excuses. Also, there are general heuristics, such as the fact that a structure is more likely to be appropriate if a desired feature is missing (perhaps because it is hidden from view) than if an inappropriate feature is present. For example, a person with one leg is more plausible than a person with a tail.
- Refer to specific stored links between structures to suggest new directions in which to explore. An example of this sort of linking among a set of frames is shown in the similarity network shown in Fig. 4.11.[5]
- If the knowledge structures are stored in an *isa* hierarchy, traverse upward in it until a structure is found that is sufficiently general that it does not conflict with the evidence. Either use this structure if it is specific enough to provide the required knowledge or consider creating a new structure just below the matching one.

[5] This example is taken from Minsky [1975].

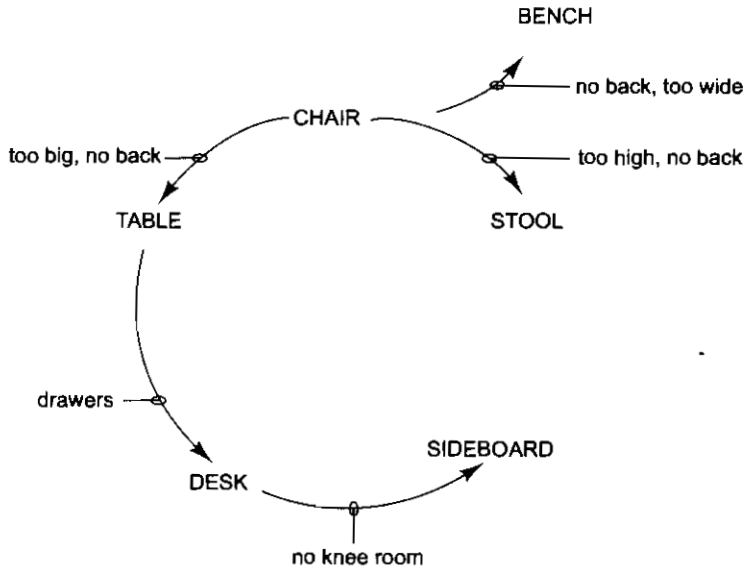


Fig. 4.11 *A Similarity Net*

4.4 THE FRAME PROBLEM

So far in this chapter, we have seen several methods for representing knowledge that would allow us to form complex state descriptions for a search program. Another issue concerns how to represent efficiently *sequences* of problem states that arise from a search process. For complex ill-structured problems, this can be a serious matter.

Consider the world of a household robot. There are many objects and relationships in the world, and a state description must somehow include facts like *on(Plant12, Table34)*, *under(Table34, Window13)*, and *in(Table34, Room 15)*. One strategy is to store each state description as a list of such facts. But what happens during the problem-solving process if each of those descriptions is very long? Most of the facts will not change from one state to another, yet each fact will be represented once at every node, and we will quickly run out of memory. Furthermore, we will spend the majority of our time creating these nodes and copying these facts—most of which do not change often—from one node to another. For example, in the robot world, we could spend a lot of time recording *above(Ceiling, Floor)* at every node. All of this is, of course, in addition to the real problem of figuring out which facts *should* be different at each node.

This whole problem of representing the facts that change as well as those that do not is known as *the frame problem* [McCarthy and Hayes, 1969]. In some domains, the only hard part is representing all the facts. In others, though, figuring out which ones change is nontrivial. For example, in the robot world, there might be a table with a plant on it under the window. Suppose we move the table to the center of the room. We must also infer that the plant is now in the center of the room too but that the window is not.

To support this kind of reasoning, some systems make use of an explicit set of axioms called *frame axioms*, which describe all the things that do not change when a particular operator is applied in state n to produce state $n + 1$. (The things that do change must be mentioned as part of the operator itself.) Thus, in the robot domain, we might write axioms such as

$$color(x,y,\ s_1) \wedge move(x,\ s_1,S_2) \rightarrow color(x,y,\ S_2)$$

which can be read as, "If x has color y in state s_1 and the operation of moving x is applied in state s_1 to produce state s_2, then the color of x in s_2 is still y." Unfortunately, in any complex domain, a huge number of these axioms becomes necessary. An alternative approach is to make the assumption that the only things that change are the things that must. By "must" here we mean that the change is either required explicitly by the axioms that describe the operator or that it follows logically from some change that is asserted explicitly. This idea of *circumscribing* the set of unusual things is a very powerful one; it can be used as a partial solution to the frame problem and as a way of reasoning with incomplete knowledge. We return to it in Chapter 7.

But now let us return briefly to the problem of representing a changing problem state. We could do it by simply starting with a description of the initial state and then making changes to that description as indicated by the rules we apply. This solves the problem of the wasted space and time involved in copying the information for each node. And it works fine until the first time the search has to backtrack. Then, unless all the changes that were made can simply be ignored (as they could be if, for example, they were simply additions of new theorems), we are faced with the problem of backing up to some earlier node. But how do we know what changes in the problem state description need to be undone? For example, what do we have to change to undo the effect of moving the table to the center of the room? There are two ways this problem can be solved:

- Do not modify the initial state description at all. At each node, store an indication of the specific changes that should be made at this node. Whenever it is necessary to refer to the description of the current problem state, look at the initial state description and also look back through all the nodes on the path from the start state to the current state. This is what we did in our solution to the cryptarithmetic problem in Section 3.5. This approach makes backtracking very easy, but it makes referring to the state description fairly complex.
- Modify the initial state description as appropriate, but also record at each node an indication of what to do to undo the move should it ever be necessary to backtrack through the node. Then, whenever it is necessary to backtrack, check each node along the way and perform the indicated operations on the state description.

Sometimes, even these solutions are not enough. We might want to remember, for example, in the robot world, that before the table was moved, it was under the window and after being moved, it was in the center of the room. This can be handled by adding to the representation of each fact a specific indication of the time at which that fact was true. This indication is called a *state variable*. But to apply the same technique to a real-world problem, we need, for example, separate facts to indicate all the times at which the Statue of Liberty is in New York.

There is no simple answer either to the question of knowledge representation or to the frame problem. Each of them is discussed in greater depth later in the context of specific problems. But it is important to keep these questions in mind when considering search strategies, since the representation of knowledge and the search process depend heavily on each other.

SUMMARY

The purpose of this chapter has been to outline the need for knowledge in reasoning programs and to survey issues that must be addressed in the design of a good knowledge representation structure. Of course, we have not covered everything. In the chapters that follow, we describe some specific representations and look at their relative strengths and weaknesses.

The following collections all contain further discussions of the fundamental issues in knowledge representation, along with specific techniques to address these issues: Bobrow [1975], Winograd [1978], Brachman and Levesque [1985], and Halpern [1986]. For especially clear discussions of specific issues on the topic of knowledge representation and use see Woods [1975] and Brachman [1985].

CHAPTER

5

USING PREDICATE LOGIC

Nature cares nothing for logic, our human logic: she has her own, which we do not recognize and do not acknowledge until we are crushed under its wheel.

—**Ivan Turgenev**
(1818–1883), Russian novelist and playwright

In this chapter, we begin exploring one particular way of representing facts — the language of logic. Other representational formalisms are discussed in later chapters. The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old — mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

One of the early domains in which AI techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics. For example, the Logic Theorist [Newell *et al.*, 1963] proved theorems from the first chapter of Whitehead and Russell's *Principia Mathematica* [1950]. Another theorem prover [Gelernter *et al.*, 1963] proved theorems in geometry. Mathematical theorem proving is still an active area of AI research. (See, for example, Wos *et al.* [1984].) But, as we show in this chapter, the usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics. It turns out that mathematics is no different from any other complex intellectual endeavor in requiring both reliable deductive mechanisms and a mass of heuristic knowledge to control what would otherwise be a completely intractable search problem.

At this point, readers who are unfamiliar with propositional and predicate logic may want to consult a good introductory logic text before reading the rest of this chapter. Readers who want a more complete and formal presentation of the material in this chapter should consult Chang and Lee [1973]. Throughout the chapter, we use the following standard logic symbols: "$\rightarrow$" (*material implication*), "$\neg$" (*not*), "$\vee$" (*or*), "$\wedge$" (*and*), "$\forall$" (*for all*), and "$\exists$" (*there exists*).

5.1 REPRESENTING SIMPLE FACTS IN LOGIC

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an AI system might need. Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists. We can easily represent real-world facts as logical *propositions* written as *well-formed formulas* (*wff's*) in propositional logic, as shown in Fig. 5.1. Using these propositions, we could, for example, conclude from the fact that it is raining the fact that it is not sunny. But very quickly we run up against the limitations of propositional logic. Suppose we want to represent the obvious fact stated by the classical sentence

It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny.
RAINING $\rightarrow \neg$ *SUNNY*

Fig. 5.1 *Some Simple Facts in Prepositional Logic*

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(*SOCRATES*)
MAN(*PLATO*)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal.

We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to first-order predicate logic (or just predicate logic, since we do not discuss higher order theories in this chapter) as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in prepositional logic. In predicate logic, we can represent real-world facts as *statements* written as wff's.

But a major motivation for choosing to use logic at all is that if we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge. Determining the validity of a proposition in propositional logic is straightforward, although it may be computationally hard. So before we adopt predicate logic as a good medium for representing knowledge, we need to ask whether it also provides a good way of reasoning with the knowledge. At first glance, the answer is yes. It provides a way of deducing new statements from old ones. Unfortunately, however, unlike propositional logic, it does not possess a decision procedure, even an exponential one. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semidecidable. A simple such procedure is to use the rules of inference to generate theorem's from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought. This method is not particularly efficient, however, and we will want to try to find a better one.

Although negative results, such as the fact that there can exist no decision procedure for predicate logic, generally have little direct effect on a science such as AI, which seeks positive methods for doing things, this particular negative result is helpful since it tells us that in our search for an efficient proof procedure, we should he content if we find one that will prove theorems, even if it is not guaranteed to halt if given a nontheorem. And the fact that there cannot exist a decision procedure that halts on all possihle inputs does not mean that there cannot exist one that will halt on almost all the inputs it would see in the process of trying to solve real problems. So despite the theoretical undecidability of predicate logic, it can still serve as a useful way of representing and manipulating some of the kinds of knowledge that an AI system might need.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.

   man(*Marcus*)

   This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge. For this simple example, it will be all right.

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x : Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

   Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.

5. All Romans were either loyal to Caesar or hated him.

   $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

   In English, the word "or" sometimes means the logical *inclusive-or* and sometimes means the logical *exclusive-or* (XOR). Here we have used the inclusive interpretation. Some people will argue, however, that this English sentence is really stating an,exclusive-or. To express that, we would have to write:

   $\forall x : Romman(x) \rightarrow [(loyal\ to(x, Caesar) \vee hate(x, Caesar)) \wedge$
   $\neg(loyalto(x, Caesar) \wedge hate(x,Caesar))]$

6. Everyone is loyal to someone.

   $\forall x : \rightarrow y : loyalto(x,y)$

   A major problem that arises when trying to convert English sentences into logical statements is the scope of quantifiers. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as $\exists y : \forall x : loyalto(x,y)$)? Often only one of the two interpretations seems likely, so people tend to favor it.

7. People only try to assassinate rulers they are not loyal to.

   $\forall x : \forall y : person(x) \wedge ruler(y) \wedge tryassassinate(x,y) \rightarrow \neg\ loyalto(x,y)$

   This sentence, too, is ambiguous. Does it mean that the only rulers that people try to assassinate are those to whom they are not loyal (the interpretation used here), or does it mean that the only thing people try to do is to assassinate rulers to whom they are not loyal?

   In representing this sentence the way we did, we have chosen to write "try to assassinate" as a single predicate. This gives a fairly simple representation with which we can reason about trying to assassinate. But using this representation, the connections between trying to assassinate and trying to do other things and between trying to assassinate and actually assassinating could not be made easily. If such connections were necessary, we would need to choose a different representation.

8. Marcus tried to assassinate Caesar.

   $tryassassinate\ (Marcus, Caesar)$

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process, see Reichenbach [1947].

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

$\neg loyalto(Marcus, Caesar)$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining. This process may require the search of an AND-OR graph (as described in Section 3.4) when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 5.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty set. The attempt fails, however, since there is no way to satisfy the goal *person (Murcus)* with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to add the representation of another fact to our system, namely:

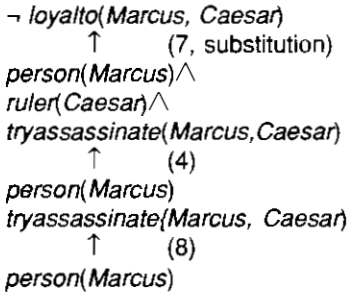


Fig. 5.2 *An Attempt to Prove ¬loyalto(Marcus,Caesar)*

9. All men are people.
   $\forall : man(x) \rightarrow person(x)$

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:

- Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
- There is often a choice of how to represent the knowledge (as discussed in connection with 1, and 7 above). Simple representations are desirable, but they may preclude certain kinds of reasoning. The expedient representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.

- Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention. We discuss this issue further in Section 10.3.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?" How would a program decide whether it should try to prove

loyalto(*Marcus, Caesar*)
¬*loyalto*(*Marcus, Caesar*)

There are several things it could do. It could abandon the strategy we have outlined of reasoning backward from a proposed truth to the axioms and instead try to reason forward and see which answer it gets to. The problem with this approach is that, in general, the branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time. A second thing it could do is use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem. Another thing it could do is simply try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty, the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

5.2 REPRESENTING INSTANCE AND *ISA* RELATIONSHIPS

In Chapter 4, we discussed the specific attributes *instance* and *isa* and described the important role they play in a particularly useful form of reasoning, property inheritance. But if we look back at the way we just represented our knowledge about Marcus and Caesar, we do not appear to have used these attributes at all. We certainly have not used predicates with those names. Why not? The answer is that although we have not used the predicates *instance* and *isa* explicitly, we have captured the relationships they are used to express, namely class membership and class inclusion.

Figure 5.3 shows the first five sentences of the last section represented in logic in three different ways. The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as *Roman*), each of which corresponds to a class. Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P. The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate *instance* is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit *isa* predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3. The implication rule there states that if an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman*. Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship. The third part contains representations that use both the *instance* and *isa* predicates explicitly. The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided. This additional axiom describes how an *instance* relation and *an isa* relation can be combined to derive a new *instance* relation. This one additional axiom is general, though, and does not need to be provided separately for additional *isa* relations.

1. *man*(*Marcus*)
2. *Pompeian*(*Marcus*)
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. *ruler*(*Caesar*)
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. *instance*(*Marcus*, *man*)
2. *instance*(*Marcus*, *Pompeian*)
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. *instance*(*Caesar*, *ruler*)
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. *instance*(*Marcus*, *man*)
2. Instance(*Marcus*, *Pompeian*)
3. *isa*(*Pompeian*, *Roman*)
4. *instance*(*Caesar*, *ruler*)
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

Fig. 5.3 *Three Ways of Representing Class Membership*

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented, those memberships need not be represented with predicates labeled *instance* and *isa*. In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

There is one additional point that needs to be made here on the subject of the use of *isa* hierarchies in logic-based systems. The reason that these hierarchies are so important is not that they permit the inference of superclass membership. It is that by permitting the inference of superclass membership, they permit the inference of other properties associated with membership in that superclass. So, for example, in our sample knowledge base it is important to be able to conclude that Marcus is a Roman because we have some relevant knowledge about Romans, namely that they either hate Caesar or are loyal to him. But recall that in the baseball example of Chapter 4, we were able to associate knowledge with superclasses that could then be overridden by more specific knowledge associated either with individual instances or with subclasses. In other words, we recorded default values that could be accessed whenever necessary. For example, there was a height associated with adult males and a different height associated with baseball players. Our procedure for manipulating the *isa* hierarchy guaranteed that we always found the correct (i.e., most specific) value for any attribute. Unfortunately, reproducing this result in logic is difficult.

Suppose, for example, that, in addition to the facts we already have, we add the following.[1]

Pompeian(*Paulus*)
$\neg$ [*loyalto*(*Paulus*, *Caesar*) $\vee$ *hate*(*Paulus*,*Caesar*)]

[1] For convenience, we now return to our original notation using unary predicates to denote class relations.

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to which an exception is being made. So our original sentence 5 must become:

$$\forall x : Roman(x) \wedge \neg eq(x,Paulus) \rightarrow loyalto(x,Caesar) \vee hate(x,Caesar)$$

In this framework, every exception to a general rule' must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there are exceptions than is the use of general rules in a semantic net.

A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance. But we defer consideration of this problem until Chapter 7.

5.3 COMPUTABLE FUNCTIONS AND PREDICATES

In the example we explored in the last section, all the simple facts were expressed as combinations of individual predicates, such as:

$$tryassassinate(Marcus,Caesar)$$

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

gt(1,0)	lt(0,1)
gt(2,1)	lt(1,2)
gt(3,2)	lt(2,3)
⋮	⋮

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*. Whatever proof procedure we use, when it comes upon one of these predicates, instead of searching for it explicitly in the database or attempting to deduce it by further reasoning, we can simply invoke a procedure, which we will specify in addition to our regular rules, that will evaluate it and return true or false.

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of

$$gt(2 + 3,1)$$

To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to *gt*.

The next example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

Consider the following set of facts, again involving Marcus:

1. Marcus was a man.

   man(Marcus)

   Again we ignore the issue of tense.
2. Marcus was a Pompeian.

   Pompeian(Marcus)

3. Marcus was born in 40 A.D.

   born(Marcus, 40)

   For simplicity, we will not represent A.D. explicitly, just as we normally omit it in everyday discussions. If we ever need to represent dates B.C., then we will have to decide on a way to do that, such as by using negative numbers. Notice that the representation of a sentence does not have to look like the sentence itself as long as there is a way to convert back and forth between them. This allows us to choose a representation, such as positive and negative numbers, that is easy for a program to work with.
4. All men are mortal.

   $\forall x$: $man(x) \rightarrow mortal(x)$

5. All Pompeians died when the volcano erupted in 79 A.D.

   $erupted(volcano, 79) \wedge \forall x : [Pompeian(x) \rightarrow died(x, 79)]$

   This sentence clearly asserts the two facts represented above. It may also assert another that we have not shown, namely that the eruption of the volcano caused the death of the Pompeians. People often assume causality between concurrent events if such causality seems plausible.

   Another problem that arises in interpreting this sentence is that of determining the referent of the phrase "the volcano." There is more than one volcano in the world. Clearly the one referred to here is Vesuvius, which is near Pompeii and erupted in 79 A.D. In general, resolving references such as these can require both a lot of reasoning and a lot of additional knowledge.
6. No mortal lives longer than 150 years.

   $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$

   There are several ways that the content of this sentence could be expressed. For example, we could introduce a function *age* and assert that its value is never greater than 150. The representation shown above is simpler, though, and it will suffice for this example.
7. It is now 1991.

   $now = 1991$

   Here we will exploit the idea of equal quantities that can be substituted for each other.

Now suppose we want to answer the question "Is Marcus alive?" A quick glance through the statements we have suggests that there may be two ways of deducing an answer. Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible. As soon as we attempt to follow

either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.

$$\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x,t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$$

This is not strictly correct, since $\neg dead$ implies alive only for animate objects. (Chairs can be neither dead nor alive.) Again, we will ignore, this for now. This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.

$$\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$$

This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall: Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall_x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t: [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Fig. 5.4 *A Set of Facts about Marcus*

To answer that requires breaking time up into smaller units than years. If we do that, we can then add rules that say such things as "One is dead at *time* (*year 1, month 1*) if one died during (*year 1, month 1*) and *month 2* precedes *month 1*]." We can extend this to days, hours, etc., as necessary. But we do not want to reduce all time statements to that level of detail, which is unnecessary and often not available.

A summary of all the facts we have now represented is given in Fig. 5.4. (The numbering is changed slightly because sentence 5 has been split into two parts.) Now let's attempt to answer the question "Is Marcus alive?" by proving:

$$\neg alive(Marcus, now)$$

Two such proofs are shown in Fig. 5.5 and 5.6. The term *nil* at the end of each proof indicates that the list of conditions remaining to be proved is empty and so the proof has succeeded. Notice in those proofs that whenever a statement of the form:

$$a \wedge b \rightarrow c$$

was used, a and b were set up as independent subgoals. In one sense they are, but in another sense they are not if they share the same bound variables, since, in that case, consistent substitutions must be made in each of them. For example, in Fig. 5.6 look at the step justified by statement 3. We can satisfy the goal

$born(Marcus, t_1)$

using statement 3 by binding $\wedge$ to 40, but then we must also bind $\wedge$ to 40 in

$gt(now - t_1, 150)$

since the two t_1's were the same variable in statement 4, from which the two goals came. A good computational proof procedure has to include both a way of determining that a match exists and a way of guaranteeing uniform substitutions throughout a proof. Mechanisms for doing both those things are discussed below.

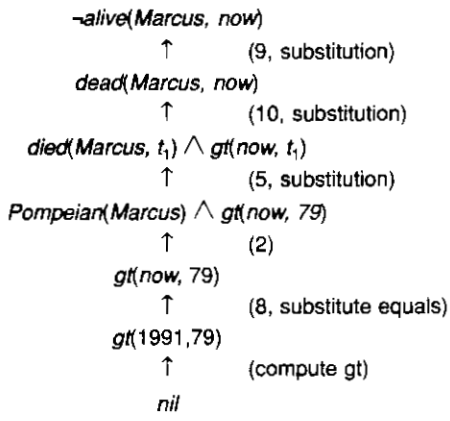


Fig. 5.5 *One Way of Proving That Marcus Is Dead*

From looking at the proofs we have just shown, two things should be clear:

- Even very simple conclusions can require many steps to prove.
- A variety of processes, such as matching, substitution, and application of *modus ponens* are involved in the production of a proof. This is true even for the simple statements we are using. It would be worse if we had implications with more than a single term on the right or with complicated expressions involving *amis* and *ors* on the left.

The first of these observations suggests that if we want to be able to do nontnvial reasoning, we are going to need some statements that allow us to take bigger steps along the way. These should represent the facts that people gradually acquire as they become experts. How to get computers to acquire them is a hard problem for which no very good answer is known.

The second observation suggests that actually building a program to do what people do in producing proofs such as these may not be easy. In the next section, we introduce a proof procedure called *resolution* that reduces some of the complexity because it operates on statements that have first been converted to a single canonical form.

5.4 RESOLUTION

As we suggest above, it would be useful from a computational point of view ff we had a proof procedure that carried out in a single operation the variety of processes involved in reasoning with statements in predicate logic. Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form, which is described below.

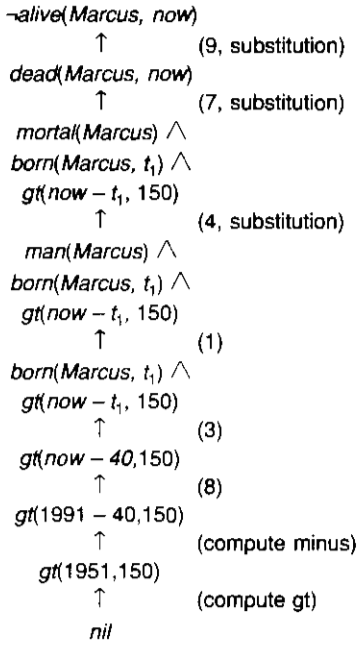


Fig. 5.6 *Another Way of Proving That Marcus is Dead*

Resolution produces proofs by *refutation.* In other words, to prove a statement (i.e., show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable). This approach contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms. Further discussion of how resolution operates will be much more straightforward after we have discussed the standard form in which statements will be represented, so we defer it until then.

5.4.1 Conversion to Clause Form

Suppose we know that all Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy. We could represent that in the following wff:

$$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$$

To use this formula in a proof requires a complex matching process. Then, having matched one piece of it, such as *thinkcrazy*(*x*,*y*), it is necessary to do the right thing with the rest of the formula including the pieces in which the matched part is embedded and those in which it is not. If the formula were in a simpler form, this proc;ess would be much easier. The formula would be easier to work with if

- It were flatter, i.e., there was less embedding of components.
- The quantifiers were separated from the rest of the formula so that they did not need to be considered.

Conjunctive normal form [Davis and Putnam, 1960] has both of these properties. For example, the formula given above for the feelings of Romans who know Marcus would be represented in conjunctive normal form as

$$\neg Roman(x) \wedge \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$$

Since there exists an algorithm for converting any wff into conjunctive normal form, we lose no generality if we employ a proof procedure (such as resolution) that operates only on wff's in this form. In fact, for resolution to work, we need to go one step further. We need to reduce a set of wff's to a set of *clauses*, where a clause is defined to be a wff in conjunctive normal form but with no instances of the connector A. We can do this by first converting each wff into conjunctive normal form and then breaking apart each such expression into clauses, one for each conjunct. All the conjuncts will be considered to be conjoined together as the proof procedure operates. To convert a wff into clause form, perform the following sequence of steps.

Algorithm: Convert to Clause Form

1. Eliminate $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$. Performing this transformation on the wff given above yields

   $\vee x : \neg[Roman(x) \wedge know< x, Marcus)] \vee$
   $[hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x,y))]$

2. Reduce the scope of each $\neg$ to a single term, using the fact that $\neg(\neg p) = p$, deMorgan's laws [which say that $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$], and the standard correspondences between quantifiers [$\neg\forall x : P(x) = \exists x : \neg P(x)$ and $\neg\exists x : P(x) = \forall x : \neg P(x)$]. Performing this transformation on the wff from step 1 yields

   $\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$
   $[hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

   $\forall x : P(x) \vee \forall x : Q(x)$

   would be converted to

   $\forall x : P(x) \vee \forall y : Q(y)$

   This step is in preparation for the next.

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of'step 2, we get

   $\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x\ Marcus)] \vee$
   $[hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x,y))]$

   At this point, the formula is in what is known as *prenex normal form*. It consists of a *prefix* of quantifiers followed by a *matrix*, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

   $\exists y : President(y)$

   can be transformed into the formula

   $President(S1)$

where SI is a function with no arguments that somehow produces a value that satisfies President. If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

$\forall x : \exists y : father\text{-}of(y,x)$

the value of y that satisfies *father-of* depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

$\forall x : father\text{-}of(S2(x),x))$

These generated functions are called *Skolem functions*. Sometimes ones with no arguments are called *Skolem constants*.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

$[\neg Roman(x) \vee \neg know(x, Marcus)] \vee$
$[hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$

7. Convert the matrix into a conjunction of disjuncts. In the case of our example, since there are no *and's*, it is only necessary to exploit the associative property of *or* [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \wedge c)$] and simply remove the parentheses, giving

$\neg Roman(x) \vee \neg know(x, Marcus) \vee$
$hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, y)$

However, it is also frequently necessary to exploit the distributive property [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]. For example, the formula

$(winter \wedge wearingboots) \vee (summer \wedge wearingsandals)$

becomes, after one application of the rule

$[winter \vee (summer \wedge wearingsandals)]$
$\wedge [wearingboots \vee (summer \wedge wearingsandals)]$

and then, after a second application, required since there are still conjuncts joined byOR's,

$(winter \vee summer) \wedge$
$(winter \vee wearingsandals) \wedge$
$(wearingboots \vee summer) \wedge$
$(wearingboots \vee wearingsandals)$

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.
9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

$$(\forall x : P(x) \wedge Q(x)) = \forall x : P(x) \wedge \forall x : Q(x)$$

Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals*. These clauses can now be exploited by the resolution procedure to generate proofs.

5.4.2 The Basis of Resolution

The resolution procedure is a simple iterative process: at each step, two clauses, called the *parent clauses*, are compared (*resolved*), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

$$winter \vee summer$$
$$\neg winter \vee cold$$

Recall that this means that both clauses must be true (i.e., the clauses, although they look independent, are really conjoined).

Now we observe that precisely one of *winter* and $\neg winter$ will be true at any point. If *winter* is true, then *cold* must be true to guarantee the truth of the second clause. If $\neg winter$ is true, then *summer* must be true to guarantee the truth of the first clause. Thus we see that from these two clauses we can deduce

$$summer \vee cold$$

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example, *winter*. The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel.

If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses

$$winter$$
$$\neg winter$$

will produce the empty clause. If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

So far, we have discussed only resolution in prepositional logic. In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables. The theoretical basis of the resolution procedure in predicate logic is Herbrand's theorem [Chang and Lee, 1973], which tells us the following:

- To show that a set of clauses S is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the *Herbrand universe* of S.
- A set of clauses S is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of S is unsatisfiable.

The second part of the theorem is important if there is to exist any computational procedure for proving unsatisfiability, since in a finite amount of time no procedure will be able to examine an infinite set. The first part suggests that one way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction. But that is highly inefficient. The resolution principle, first introduced by Robinson [1965], provides a way of finding contradictions by trying a minimum number of substitutions. The idea is-to keep clauses in their general form as long as possible and only introduce specific substitutions when they are required. For more details on different kinds of resolution, see Stickel [1988].

5.4.3 Resolution in Propositional Logic

In order to make it clear how resolution works, we first present the resolution procedure for propositional logic. We then expand it to include predicate logic.

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

Algorithm: Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resulting clause, called the *resolvent,* will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Fig. 5.7 and we want to prove R. First we convert the axioms to clause form, as shown in the second column of the figure.

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Fig. 5.7 *A Few Facts in Propositional Logic*

Then we negate R, producing $\neg R$, which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause (shown as a box). We might, for example, generate the sequence of resolvents shown in Fig. 5.8. We begin by resolving with the clause $\neg R$ since that is one of the clauses that must be involved in the contradiction we are trying to find.

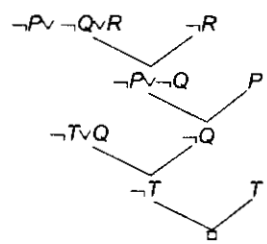


Fig. 5.8 *Resolution in Propositional Logic*

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause. To see how this works, let's look again at the example. In order for proposition 2 to be true., one of three things must be true: $\neg P$, $\neg Q$, or R. But we are assuming that $\neg R$ is true. Given that, the only way for proposition 2 to be true is for one of two things to be true: $\neg P$ or $\neg Q$. That is what the first resolvent clause says. But proposition 1 says that P is true, which means that $\neg P$ cannot be true, which leaves only one way for proposition 2 to be true, namely for $\neg Q$ to be true (as shown in the second resolvent clause). Proposition 4 can be true if either $\neg T$ or Q is true. But since we now know that $\neg Q$ must be true, the only way for proposition 4 to be true is for $\neg T$ to be true (the third resolvent). But proposition 5 says that T is true. Thus there is no way for all of these clauses to be true in a single interpretation. This is indicated by the empty clause (the last resolvent).

5.4.4 The Unification Algorithm

In propositional logic, it is easy to determine that two literals cannot both be true at the same time. Simply look for L and $\neg L$ In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For example, *man*(*John*) and ¬*man*(*John*) is a contradiction, while *man*(*John*) artd ¬*man*(*Spot*) is not Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the *unification algorithm,* that does just this.

The basic idea of unification is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For example, the two literals

tryassassinate(*Marcus, Caesar*)
hate(*Marcus, Caesar*)

cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose we want to unify the expressions

$P(x,x)$
$P(y,z)$

The two instances of P match fine. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write that substitution as

y/x

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match x and z. we produce tne substitution z/x. But we cannot substitute bothy and z for x, so we have not produced a consistent substitution.

What we need to do after finding the first substitutiony/x is to make that substitution throughout the literals, giving

P(*y*, *y*)
P(*y*, *z*)

Now we can attempt to unify arguments v and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

$(z/y)(y/x)$

following standard notation for function composition. In general, the substitution $(a_1/a_2, a_3/a_4, ...)(b_1/b_2, b_3/b_4,...)...$ means to apply all the substitutions of the right- most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. ' For example, the literals

hate(*x*, *y*)
hate(*Marcus*, z)

could be unified with any of the following substitutions:

(*Marcus*/*x*,*z*/*y*)
(*Marcus*/*x*,*y*/*z*)
(*Marcus*/*x*,*Caesar*/*y*,*Caesar*/*z*)
(*Marcus*/*x*,*Polonius*/*y*,*Polonius*/*z*)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify(Ll, L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was fpund without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(Ll, L2)

1. If *L*1 or *L*2 are both variables or constants, then:
   (a) If *L*1 and *L*2 are identical, then return NIL.
   (b) Else if *L*1 is a variable, then if *L*1 occurs in *L*2 then return {FAIL}, else return (*L*2/*L*1).
   (c) Else if *L*2 is a variable then if *L*2 occurs in *L*1 then return {FAIL}, else return (*L*1/*L*2).
   (d) Else return {FAIL}.

2. If the initial predicate symbols in $L1$ and $L2$ are not identical, then return {FAIL).
3. If LI and $L2$ have a different number of arguments, then return {FAIL}.
4. Set *SUBST* to NIL. (At the end of this procedure, *SUBST* will contain all the substitutions used to unify $L1$ and $L2$.)
5. For $i \leftarrow 1$ to number of arguments in $L1$:
   (a) Call Unify with the /th argument of $L1$ and the ith argument of $L2$, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
      (i) Apply S to the remainder of both $L1$ and $L2$.
      (ii) *SUBST* : = APPEND(S, *SUBST*).
6. Return *SUBST*.

The only part of this algorithm that we have not yet discussed is the check in steps 1(b) and l(c) to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

$f(x,x)$
$f(g(x),g(x))$

If we accepted $g(x)$ as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.

Unification has deep mathematical roots and is a useful operation in many AI programs, for example, theorem provers and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed. For an introduction to these techniques and applications, see Knight [1989].

5.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, *man*(x) and ¬*man*(*Spot*) are contradictory, since *man*(x) and *man*(*Spot*) can be unified. This corresponds to the intuition that says that *man*(x) cannot be true for all x if there is known to be some x, say Spot, for which *man*(x) is false. Thus in order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. *man*(*Marcus*)
2. ¬*man*(x_1) $\vee$ *mortal*(x_1)

The literal *man{Marcus*) can be unified with the literal *man{x\)* with the substitution *Marcus*/x_1, telling us that for x_1 = *Marcus*, ¬*man*(*Marcus*) is false. But we cannot simply cancel out the two *man* literals as we did in propositional logic and generate the resolvent *mortal*(x_1). Clause 2 says that for a given x_1, either ¬*man*(x_1) or *mortal*(x_1). So for it to be true, we can now conclude only that *mortal*(*Marcus*) must be true. It is not necessary that *mortal*(x_1) be true for all x_1, since for some values of x_1, ¬*man*(x_1) might be true, making *mortal*(x_1) irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be *mortal*(*Marcus*), which we get by applying the result of the unification process to.the resolvent. The resolution process can then proceed from there to discover whether *mortal*(*Marcus*) leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has bee'n done, it is easy to determine how the

unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg T2$ such that one of the parent clauses contains $T2$ and the other contains $T1$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $T2$ should appear in the resolvent. We call $T1$ and $T2$ *Complementary literals.* Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

- Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predi- cates can be located directly.
- Eliminate certain clauses as soon as they are generated so that they cannot partic- ipate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed by other clauses (i.e., they are easier ;o satisfy. For example, $P \vee Q$ is subsumed by P.)
- Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.
- Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy.*

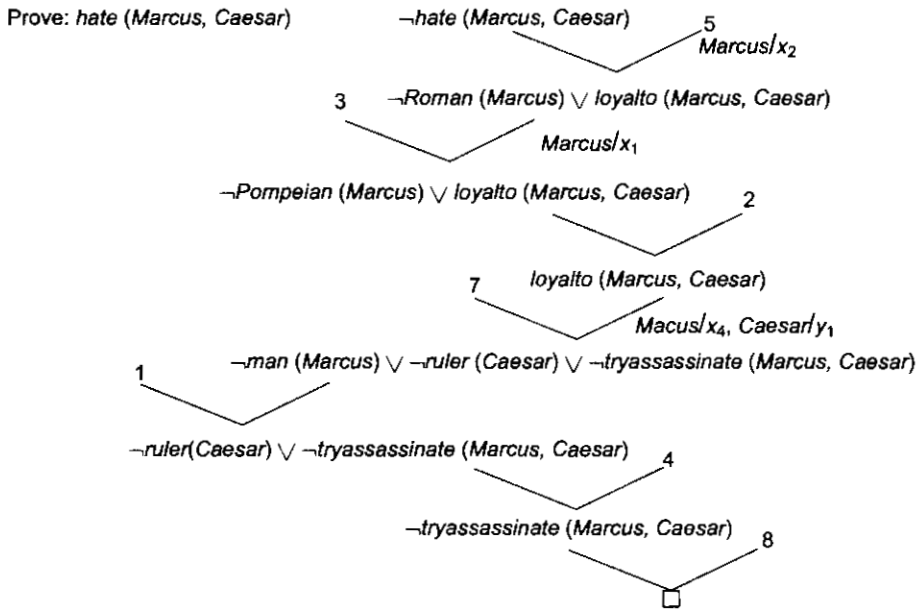
Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 5.1. To use them in resolution proofs, we must convert them to clause form as described in Section 5.4.1. Figure 5.9(a) shows the results of that conversion. Figure 5.9(b) shows a resolution proof of the statement

hate(Marcus,Caesar)

Axioms in clause form:

1. *man*(*Marcus*)
2. *Pompeian*(*Marcus*)
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. *ruler*(*Caesar*)
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. *tryassassinate*(*Marcus*,*Caesar*)

(a)



(b)

Fig. 5.9 *A Resolution Proof*

Of course, many more resolvents could have been generated than we have shown, but we used the heuristics described above to guide the search. Notice that what we have done here essentially is to reason backward from the statement we want to show is a contradiction through a set of intermediate conclusions to the final conclusion of inconsistency.

Suppose our actual goal in proving the assertion

hate(*Marcus*,*Caesar*)

was to answer the question "Did Marcus hate Caesar?" In that case, we might just as easily have attempted to prove the statement

¬*hate*(*Marcus*,*Caesar*)

To do so, we would have added

hate(Marcus, Caesar)

to the set of available clauses and begun the resolution process. But immediately we notice that there are no clauses that contain a literal involving ¬*hate*. Since the resolution process can only generate new clauses that are composed of combinations of literals from already existing clauses, we know that no such clause can be generated and thus we conclude that *hate(Marcus,Caesar)* will not produce a contradiction with the known statements. This is an example of the kind of situation in which the resolution procedure can detect that no contradiction exists. Sometimes this situation is detected not at the beginning of a proof, but part way through, as shown in the example in Figure 5.10(*a*), based on the axioms given in Fig. 5.9.

But suppose our knowledge base contained the two additional statements

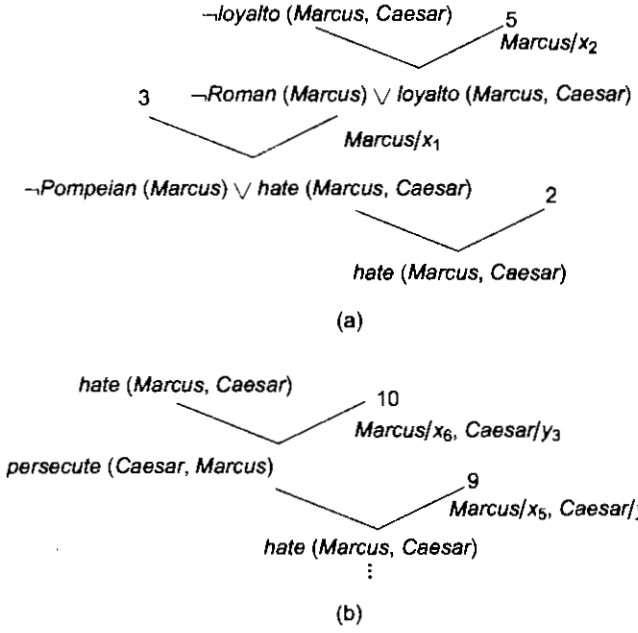


Fig. 5.10 *An Unsuccessful Attempt at Resolution*

9. $persecute(x, y) \rightarrow hate(y, x)$
10. $hate(x, y) \rightarrow persecute(y, x)$

Converting to clause form, we get

9. $\neg persecute(x_5, y_2) \vee hate(y_2, x_5)$
10. $\neg hate(x_6, y_3) \vee persecute(y_3, x_6)$

These statements enable the proof of Fig. 5.10(*a*) to continue as shown in Fig. 5.10(*b*). Now to detect that there is no contradiction we must discover that the only resolvents that can be generated have been generated before. In other words, although we can generate resolvents, we can generate no new ones.

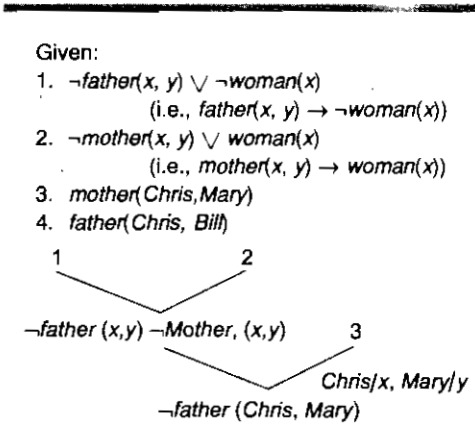


Fig. 5.11 *The Need to Standardize Variables*

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg morta(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 2008$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

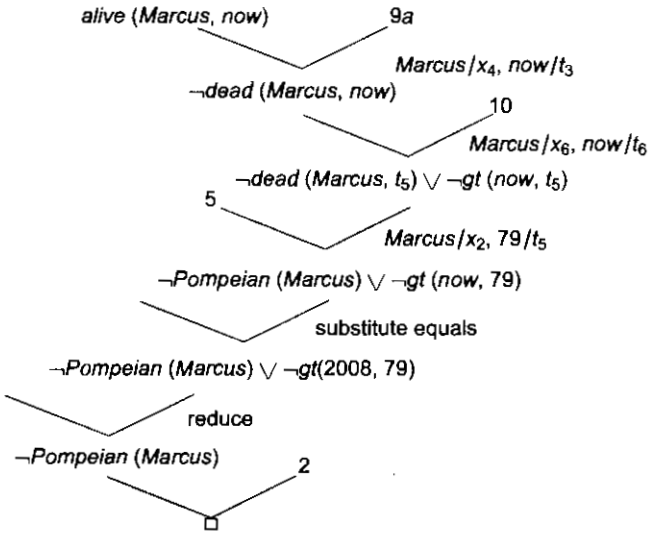


Fig. 5.12 *Using Resolution with Equality and Reduce*

Recall that the final step of the process of converting a set of formulas to clause form was to standardize apart the variables that appear in the final clauses. Now that we have discussed the resolution procedure, we can see clearly why this step is so important. Figure 5.11 shows an example of the difficulty that may arise if standardization is not done. Because the variable y occurs in both clause 1 and clause 2, the substitution at the second resolution step produces a clause that is too restricted and so does not lead to the contradiction that is present in the database. If, instead, the clause

$$\neg father(Chris, y)$$

had been produced, the contradiction with clause 4 would have emerged. This would have happened if clause 2 had been rewritten as

$$\neg mother(a, b) \vee woman(a)$$

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as we pointed out in Section 5.3, it is often more efficient to represent certain kinds of information in the form of computable functions, computable predicates, and equality relationships. It is not hard to augment resolution to handle this sort of knowledge. Figure 5.12 shows, a resolution proof of the statement

$$\neg alive(Marcus,now)$$

based on the statements given in Section 5.3. We have added two ways of generating new clauses, in addition to the resolution rule:

- Substitution of one value for another to which it is equal.
- Reduction of computable predicates. If the predicate evaluates to FALSE, it can simply be dropped, since adding V FALSE to a disjunction cannot change its truth value. If the predicate evaluates to TRUE, then the generated clause is a tautology and cannot lead to a contradiction.

5.4.6 The Need to Try Several Substitutions

Resolution provides a very good way of finding a refutation proof without actually trying all the substitutions that Herbrand's theorem suggests might be necessary. But it does not always eliminate the necessity of trying more than one substitution. For example, suppose we know, in addition to the statements in Section 5.1, that

$$hate(Marcus, Paulus)$$
$$hate(Marcus, Julian)$$

Now if we want to prove that Marcus hates some ruler, we would be likely to try each substitution shown in Figure 5.13(a) and (b) before finding the contradiction shown in (c). Sometimes there is no way short of very good luck to avoid trying several substitutions.

5.4.7 Question Answering

Very early in the history of AI it was realized that theorem-proving techniques could be applied to the problem of answering questions. As we have already suggested, this seems natural since both deriving theorems from axioms and deriving new facts (answers) from old facts employ the process of deduction. We have already shown how resolution can be used to answer yes-no questions, such as "Is Marcus alive?" In this section, we show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or

"Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the terms given in the question and then responding with another piece of that same statement that fills the slot demanded by the question. For example, to answer the question "When did Marcus die?" we need a statement of the form

died(Marcus, ??)

with ?? actually filled in by some particular year. So, since we can prove the statement

died(Marcus, 79)

we can respond with the answer 79.

It turns out that the resolution procedure provides an easy way of locating just the statement we need and finding a proof for it. Let's continue with the example question

Prove: $\exists x : hate(Marcus,x) \wedge ruler(x)$
(negate): $\neg\exists x : hate(Marcus,x) \wedge ruler(x)$
(clausify): $\neg hate(Marcus,x) \vee \neg ruler(x)$

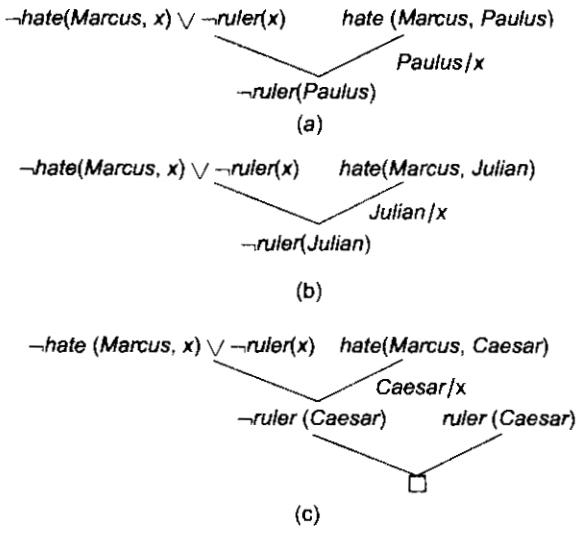


Fig. 5.13 *Trying Several Substitutions*

'When did Marcus die?" In order to be able to answer this question, it must first be true that Marcus died. Thus it must be the case that

$\exists t : died(Marcus,t)$

A reasonable first step then might be to try to prove this. To do so using resolution, we attempt to show that

$\neg\exists t$: *died(Marcus, i)*

produces a contradiction. What does it mean for that statement to produce a contradiction? Either it conflicts with a statement of the form

$\forall t$: *died(Marcus, t)*

where t is a variable, in which case we can either answer the question by reporting that there are many times at which Marcus died, or we can simply pick one such time and respond with it. The other possibility is that we produce a contradiction with one or more specific statements of the form

died(Marcus, date)

for some specific value of *date*. Whatever value of date we use in producing that contradiction is the answer we want. The value that proves that there is a value (and thus the inconsistency of the statement that there is no such value) is exactly the value we want.

Figure 5.14(a) shows how the resolution process finds the statement for which we are looking. The answer to the question can then be derived from the chain of unifications that lead back to the starting clause. We can eliminate the necessity for this final step by adding an additional expression to the one we are going to use to try to find a contradiction. This new expression will simply be the one we are trying to prove true (i.e., it will be the negation of the expression that is actually used in the resolution). We can tag it with a special marker so that it will not interfere with the resolution process. (In the figure, it is underlined.) It will just get carried along, but each time unification is done, the variables in this dummy expression will be bound just as are the ones in the clauses that are actively being used. Instead of terminating on reaching the nil clause, the resolution procedure will terminate when all that is left is the dummy expression. The bindings of its variables at that point provide the answer to the question. Figure 5.14(fr) shows how this process produces an answer to our question.

Unfortunately, given a particular representation of the facts in a system, there will usually be some questions that cannot be answered using this mechanism. For example, suppose that we want to answer the question "What happened in 79 A.D.?" \ising the statements in Section 5.3. In order to answer the question, we need to prove that something happened in 79. We need to prove

$\exists x : event(x, 79)$

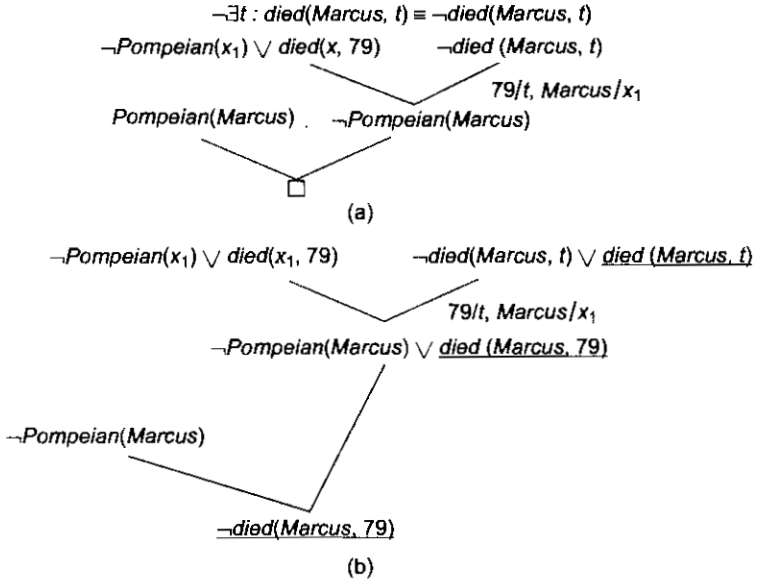


Fig. 5.14 *Answer Extraction Using Resolution*

and to discover a value for x. But we do not have any statements of the form $event(x, y)$.

We can, however, answer the question if we change our representation. Instead of saying

$$erupted(volcano, 79)$$

we can say

$$event(erupted(volcano), 79)$$

Then the simple proof shown in Fig. 5.15 enables us to answer the question.

This new representation has the drawback that it is more complex than the old one. And it still does not make it possible to answer all conceivable questions. In general, it is necessary to decide on the kinds of questions that will be asked and to design a representation appropriate for those questions.

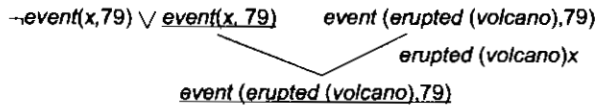


Fig. 5.15 *Using the New Representation*

Of course, yes-no and fill-in-the-blank questions are not the only kinds one could ask. For example, we might ask how to do something. So we have not yet completely solved the problem of question answering. In later chapters, we discuss some other methods for answering a variety of questions. Some of them exploit resolution; others do not.

5.5 NATURAL DEDUCTION

In the last section, we introduced resolution as an easily implementable proof procedure that relies for its simplicity on a uniform representation of the statements it uses. Unfortunately, uniformity has its price—everything looks the same. Since everything looks the same, there is no easy way to select those statements that are the most likely to be useful in solving a particular problem. In converting everything to clause form, we often lose valuable heuristic information that is contained in the original representation of the facts. For example, suppose we believe that all judges who are not crooked are well-educated, which can be represented as

$$\forall x : judge(x) \wedge \neg crooked(x) \rightarrow educated(x)$$

In this form, the statement suggests a way of deducing that someone is educated. But when the same statement is converted to clause form,

$$\neg judge(x) \vee crooked(x) \vee educated(x)$$

it appears also to be a way of deducing that someone is not a judge by showing that he is not crooked and not educated. Of course, in a logical sense, it is. But it is almost certainly not the best way, or even a very good way, to go about showing that someone is not a judge. The heuristic information contained in the original statement has been lost in the transformation.

Another problem with the use of resolution as the basis of a theorem-proving system is that people do not think in resolution. Thus it is very difficult for a person to interact with a resolution theorem prover, either to

give it advice or to be given advice by it. Since proving very hard things is something that computers still do poorly, it is important from a practical standpoint that such interaction be possible. To facilitate it, we are foiced to look for a way of doing machine theorem proving that corresponds more closely to the processes used in human theorem proving. We are thus led to what we call, mostly by definition, *natural deduction.*

Natural deduction is not a precise term. Rather it describes a melange of techniques, used in combination to solve problems that are not tractable by any one method alone. One common technique is to arrange knowledge, not by predicates, as we have been doing, but rather by the objects involved in the predicates. Some techniques for doing this are described in Chapter 9. Another technique is to use a set of rewrite rules that not only describe logical implications but also suggest the way that those implications can be exploited in proofs.

For a good survey of the variety of techniques that can be exploited in a natural deduction system, see Bledsoe [1977]. Although the emphasis in that paper is on proving mathematical theorems, many of the ideas in it can be applied to a variety of domains in which it is necessary to deduce new statements from known ones. For another discussion of theorem proving using natural mechanisms, see Boyer and Moore [1988], which describes a system for reasoning about programs. It places particular emphasis on the use of mathematical induction as a proof technique.

SUMMARY

In this chapter we showed how predicate logic can be used as the basis of a technique for knowledge representation. We also discussed a problem-solving technique, resolution, that can be applied when knowledge is represented in this way. The resolution procedure is not guaranteed to halt if given a nontheorem to prove. But is it guaranteed io halt and find a contradiction if one exists? This is called the *completeness* question. In the form in which we have presented the algorithm, the answer to this question is no. Some small changes, usually not implemented in theorem-proving systems, must be made to guarantee completeness. But, from a computational point of view, completeness is not the only important question. Instead, we must ask whether a proof can be found in the limited amount of time that is available. There are two ways to approach achieving this computational goal. The first is to search for good heuristics that can inform a theorem-proving program. Current theorem-proving research attempts to do this. The other approach is to change not the program but the data given to the program. In this approach, we recognize that a knowledge base that is just a list of logical assertions possesses no structure. Suppose an information-bearing structure could be imposed on such a knowledge base. Then that additional information could be used to guide the program that uses the knowledge. Such a program majuiot look a lot like a theorem prover, although it will still be a knowledge-based problem solver. We discuss this idea further in Chapter 9.

A second difficulty with the use of theorem proving in AI systems is that there are some kinds of information that are not easily represented in predicate logic. Consider the following examples:

- "It is very hot today." How can relative degrees of heat be represented?
- "Blond-haired people often have blue eyes." How can the amount of certainty be represented?
- "If there is no evidence to the contrary, assume that any adult you meet knows how to read." How can we represent that one fact should be inferred from the absence of another?
- "It's better to have more pieces on the board than the opponent has." How can we represent this kind of heuristic information?
- "I know Bill thinks the Giants will win, but I think they are going to lose." How can several different

belief systems be represented at once?

These examples suggest issues in knowledge representation that we have not yet satisfactorily addressed. They deal primarily with the need to make do with a knowledge base that is incomplete, although other problems also exist, such as the difficulty of representing continuous phenomena in a discrete system. Some solutions to these problems are presented in the remaining chapters in this part of the book.

EXERCISES

1. Using facts 1–9 of Section 5.1, answer the question, "Did Marcus hate Caesar?"
2. In Section 5.3, we showed that given our facts, there were two ways to prove the statement $\neg alive(Marcus, now)$. In Fig. 5.12(a) resolution proof corresponding to one of those methods is shown. Use resolution to derive another proof of the statement using the other chain of reasoning.
3. Trace the operation of the unification algorithm on each of the following pairs of literals:
   (a) $f(Marcus)$ and $f(Caesar)$
   (b) $f(x)$ $mdf(g(y))$
   (c) $f(Marcus, g(x, y))$ and$f(x, g(Caesar, Marcus))$
4. Consider the following sentences:
   - John likes all kinds of food.
   - Apples are food.
   - Chicken is food.
   - Anything anyone eats and isn't killed by is food.
   - Bill eats peanuts and is still alive.
   - Sue eats everything Bill eats.

   (a) Translate these sentences into formulas in predicate logic.
   (b) Prove that John likes peanuts using backward chaining.
   (c) Convert the formulas of part a into clause form.
   (d) Prove that John likes peanuts using resolution.
   (e) Use resolution to answer the question, "What food does Sue eat?"
5. Consider the following facts:
   - The members of the Elm St. Bridge Club are Joe, Sally, Bill, and Ellen.
   - Joe is married to Sally.
   - Bill is Ellen's brother.
   - The spouse of every married person in the club is also in the club.
   - The last meeting of the club was at Joe's house.

   (a) Represent these facts in predicate logic.
   (b) From the facts given above, most people would be able to decide on the truth of the following additional statements:
   - The last meeting of the club was at Sally's house.
   - Ellen is not married.

   Can you construct resolution proofs to demonstrate the truth of each of these statements given the five facts listed above? Do so if possible. Otherwise, add the facts you need and then construct the proofs.
6. Assume the following facts:
   - Steve only likes easy courses.
   - Science courses are hard.

- All the courses in the basketweaving department are easy.
- BK301 is a basketweaving course.

Use resolution to answer the question, "What course would Steve like?"

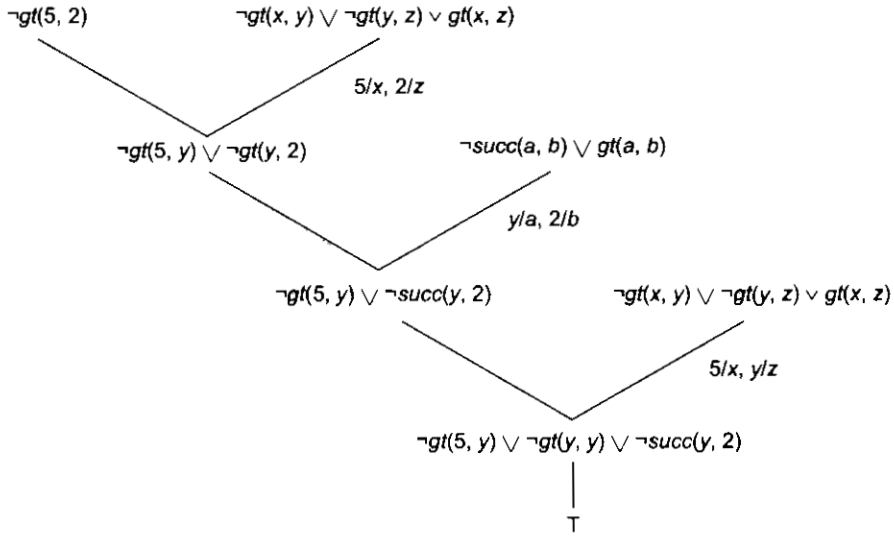
7. In Section 5.4.7, we answered the question, "When did Marcus die?" by using resolution to show that there was a time when Marcus died. Using the facts given in Fig. 5.4, and the additional fact
$\forall x : \forall t_1 : dead(x, t_1) \rightarrow \exists t_2 : gt(t_1, t_2) \wedge died(x, t_2)$
there is another way to show that there was a time when Marcus died.
   (a) Do a resolution proof of this other chain of reasoning.
   (b) What answer will this proof give to the question, "When did Marcus die?"
8. Suppose that we are attempting to resolve the following clauses:

   loves(father(a), a)
   $\neg loves(y, x) \vee loves(x, y)$

   (a) What will be the result of the unification algorithm when applied to clause 1 and the first term of clause 2?
   (b) What must be generated as a result of resolving these two clauses?
   (c) What does this example show about the order in which the substitutions determined by the unification procedure must be performed?
9. Suppose you are given the following facts:
   $\forall x, y, z : gt(x, y) \wedge gt(y, z) \rightarrow gt(x, z)$
   $\forall a,b : succ(a, b) \rightarrow gt(a,b)$
   $\forall x : \neg gt(x,x)$
   You want to prove that

   $gt(5,2)$

   Consider the following attempt at a resolution proof:



   (a) What went wrong?
   (b) What needs to be added to the resolution procedure to make sure that this does not happen?

10. The answer to the last problem suggests that the unification procedure could be simplified by omitting the check that prevents x and $f(x)$ from being unified together (the *occur check*). This should be possible since no two clauses will ever share variables. If x occurs in one, $f(x)$ cannot occur in another. But suppose the unification procedure is given the following two clauses (in the notation of Section 5.4.4):
$p(x,f(x))$
$p(f(a),a)$
Trace the execution of the procedure. What does this example show about the need for the occur check?
11. What is wrong with the following argument [Henle, 1965]?
    - Men are widely distributed over the earth.
    - Socrates is a man.
    - Therefore, Socrates is widely distributed over the earth.

    How should the facts represented by these sentences be represented in logic so that this problem does not arise?
12. Consider all the facts about baseball that are represented in the slot-and-filler structure of Fig. 4.5. Represent those same facts as a set of assertions in predicate logic. Show how the inferences that were derived from that knowledge" in Section 4.2 can be derived using logical deduction.
13. What problems would be encountered in attempting to represent the following statements in predicate logic? It should be possible to deduce the final statement from the others.
    - John only likes to see French movies.
    - It's safe to assume a movie is American unless explicitly told otherwise.
    - The Playhouse rarely shows foreign films.
    - People don't do things that will cause them to be in situations that they don't like.
    - John doesn't go to the Playhouse very often.

CHAPTER

6

REPRESENTING KNOWLEDGE USING RULES

To be useful, a system has to do more than just correctly perform some task.

—**John McDermott,**
AI Researcher

In this chapter, we discuss the use of rules to encode knowledge. This is a particularly important issue since rule-based reasoning systems have played a very important role in the evolution of AI from a purely laboratory science into a commercially significant one, as we see later in Chapter 20.

We have already talked about rules as the basis for a search program. But we gave little consideration to the way knowledge about the world was represented in the rules (although we can see a simple example of this in Section 4.2). In particular, we have been assuming that search control knowledge was maintained completely separately from the rules themselves. We will now relax that assumption and consider a set of rules to represent both knowledge about relationships in the world, as well as knowledge about how to solve problems using the content of the rules.

6.1 PROCEDURAL VERSUS DECLARATIVE KNOWLEDGE

Since our discussion of knowledge representation has concentrated so far on the use of logical assertions, we use logic as a starting point in our discussion of rule-based systems.

In the previous chapter, we viewed logical assertions as declarative representations of knowledge. A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. To use a declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how. For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a *program*, rather than as *data* to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths. These reasoning paths define the possible execution paths of the program in much the same way that traditional control constructs, such as *if-then-else*, define the execution paths through traditional programs. In other words, we could view logical assertions as

procedural representations of knowledge. A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Actually, viewing logical assertions as code is not a very radical idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

$man(Marcus)$
$man(Caesar)$
$person(Cleopatra)$
$\forall x : man(x) \rightarrow person(x)$

Now consider trying to extract from this knowledge base the answer to the question

$\exists y : person(y)$

We want to bind y to a particular value for which *person* is true. Our knowledge base justifies any of the following answers:

$y = Marcus$
$y = Caesar$
$y = Cleopatra$

Because there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response. If we view the assertions as declarative, then they do not themselves say anything about how they will be examined. If we view them as procedural, then they do. Of course, nondeterministic programs are possible — for example, the concurrent and parallel programming constructs described in Dijkstra [1976], Hoare [1985], and Chandy and Misra [1989]. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined. There is no hardware support for randomness. So if the interpreter must have a way of deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

$y = Cleopatra$

To see clearly the difference between declarative and procedural representations, consider the following assertions:

$man(Marcus)$
$man(Caesar)$
$\forall x : man(x) \rightarrow person(x)$
$person(Cleopatra)$

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed procedurally, and using the control model we used to get *Cleopatra* as our answer before, this is a different knowledge base since now the answer to our question is *Marcus*. This happens because the first statement that can achieve the *person* goal is the inference rule $\forall x : man(x) \rightarrow person(x)$. This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now *Marcus* is found to satisfy the subgoal and thus also the goal. So *Marcus* is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the *person* question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report *Caesar* as our answer.

There has been a great deal of controversy in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clearcut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called *logic programming*, and then we consider more flexible structures for rule-based systems.

6.2 LOGIC PROGRAMMING

Logic programming is a programming language paradigm in which logical assertions are viewed as programs, as described in the previous section. There are several logic programming systems in use today, the most popular of which is PROLOG [Clocksin and Mellish, 1984; Bratko, 1986]. Programming in PROLOG has been described in more detail in Chapter 25. A PROLOG program is described as a series of logical assertions, each of which is a *Horn clause*.[1] A Horn clause is a clause (as defined in Section 5.4.1) that has at most one positive literal. Thus p, $\neg p \vee q$, and $p \rightarrow q$ are all Horn clauses. The last of these does not look like a clause

$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \vee dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$
$poodle(ftujfy)$

A Representation in Logic

```
apartmentpet(X) :- pet(X), small(X).
pet(X) :- cat(X).
pet(X) :- dog(X).
dog(X) :- poodle (X) .
small(X) :- poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

Fig. 6.1 *A Declarative and a Procedural Representation*

[1] Programs written in pure PROLOG are composed only of Horn clauses. PROLOG, as an actual programming language, however, allows departures from Horn clauses. In the rest of this section, we limit our discussion to pure PROLOG.

and it appears to have two positive literals. But recall from Section 5.4.1 that any logical expression can be converted to clause form. If we do that for this example, the resulting clause is $\neg p \vee q$, which is a well-formed Horn clause. As we will see below, when Horn clauses are written in PROLOG programs, they actually look more like the form we started with (an implication with at most one literal on the right of the implication sign) than the clause form we just produced. Some examples of PROLOG Horn clauses appear below.

The fact that PROLOG programs are composed only of Horn clauses and not of arbitrary logical expressions has two important consequences. The first is that because of the uniform representation a simple and efficient interpreter can be written. The second consequence is even more important. The logic of Horn clause systems is decidable (unlike that of full first-order predicate logic).

The control structure that is imposed on a PROLOG program by the PROLOG interpreter is the same one we used at the beginning of this chapter to find the answers *Cleopatra* and *Marcus*. The input to a program is a goal to be proved. Backward reasoning is applied to try to prove the goal given the assertions in the program. The program is read top to bottom, left to right and search is performed depth-first with backtracking.

Figure 6.1 shows an example of a simple knowledge base represented in standard logical notation and then in PROLOG. Both of these representations contain two types of statements, *facts,* which contain only constants (i.e., no variables) and *rules,* which do contain variables. Facts represent statements about specific objects. Rules represent statements about classes of objects.

Notice that there are several superficial, syntactic differences between the logic and the PROLOG representations, including:

1. In logic, variables are explicitly quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted (see below). The distinction between variables and constants is made in PROLOG by having all variables begin with upper case letters and all constants begin with lower case letters or numbers.
2. In logic, there are explicit symbols for *and* ($\wedge$) and *or* ($\vee$). In PROLOG, there is an explicit symbol for *and* (,), but there is none for *or.* Instead, disjunction must be represented as a list of alternative statements, any one of which may provide the basis for a conclusion.
3. In logic, implications of the form "p implies q" are written as $p \rightarrow q$. In PROLOG, the same implication is written "backward," as `q :- p`. This form is natural in PROLOG because the interpreter always works backwards from a goal, and this form causes every rule to begin with the component that must therefore be matched first. This first component is called the *head* of the rule.

The first two of these differences arise naturally from the fact that PROLOG programs are actually sets of Horn clauses that have been transformed as follows:

1. If the Horn clause contains no negative literals (i.e., it contains a single literal which is positive), then leave it as it is.
2. Otherwise, rewrite the Horn clause as an implication, combining all of the negative literals into the antecedent of the implication and leaving the single positive literal (if there is one) as the consequent.

This procedure causes a clause, which originally consisted of a disjunction of literals (all but one of which were negative), to be transformed into a single implication whose antecedent is a conjunction of (what are now positive) literals. Further, recall that in a clause, all variables are implicitly universally quantified. But, when we apply this transformation (which essentially inverts several steps of the procedure we gave in Section 5.4.1 for converting to clause form), any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent (the head) are still universally quantified. For example, the PROLOG clause

```
P(x) :- Q(x, y)
```

is equivalent to the logical expression

$$\forall x : \exists y : Q(x, y) \rightarrow P(x)$$

A key difference between logic and the PROLOG representation is that the PROLOG interpreter has a fixed control strategy, and so the assertions in the PROLOG program define a particular search path to an answer to any question. In contrast, the logical assertions define only the set of answers that they justify; they themselves say nothing about how to choose among those answers if there are more than one.

The basic PROLOG control strategy outlined above is simple. Begin with a problem statement, which is viewed as a goal to be proved. Look for assertions that can prove the goal. Consider facts, which prove the goal directly, and also consider any rule whose head matches the goal. To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure (recall Section 5.4.4). Reason backward from that goal until a path is found that terminates with assertions in the program. Consider paths using a depth-first search strategy and using backtracking. At each choice point, consider options in the order in which they appear in the program. If a goal has more than one conjunctive part, prove the parts in the order in which they appear, propagating variable bindings as they are determined during unification. We can illustrate this strategy with a simple example.

Suppose the problem we are given is to find a value of X that satisfies the predicate `apartmentpet(X)`. We state this goal to PROLOG as

```
?- apartmentpet(X).
```

Think of this as the input to the program. The PROLOG interpreter begins looking for a fact with the predicate `apartmentpet` or a rule with that predicate as its head. Usually PROLOG programs are written with the facts containing a given predicate coming before the rules for that predicate so that the facts can be used immediately if they are appropriate and the rules will only be used when the desired fact is not immediately available. In this example, there are no facts with this predicate, though, so the one rule there is must be used. Since the rule will succeed if both of the clauses on its right-hand side can be satisfied, the next thing the interpreter does is to try to prove each of them. They will be tried in the order in which they appear. There are no facts with the predicate `pet` but again there are rules with it on the right-hand side. But this time there are two such rules, rather than one. All that is necessary for a proof though is that one of them succeed. They will be tried in the order in which they occur. The first will fail because there are no assertions about the predicate cat in the program. The second will eventually lead to success, using the rule about dogs and poodles and using the fact `poodle(fluffy)`. This results in the variable X being bound to `fluffy`. Now the second clause `small (X)` of the initial rule must be checked. Since X is now bound to fluffy, the more specific goal, small (fluffy), must be proved. This too can be done by reasoning backward to the assertion `poodle (fluffy)`. The program then halts with the result `apartmentpet(fluffy)`.

Logical negation ($\neg$) cannot be represented explicitly in pure PROLOG. So, for example, it is not possible to encode directly the logical assertion

$$\forall x : dog(x) \rightarrow \neg cat(x)$$

Instead, negation is represented implicitly by the lack of an assertion. This leads to the problem-solving strategy called *negation as failure* [Clark, 1978]. If the PROLOG program of Fig. 6.1 were given the goal

```
?- cat(fluffy).
```

it would return FALSE because it is unable to prove that Fluffy is a cat. Unfortunately, this program returns the same answer when given the goal even though the program knows nothing about Mittens and specifically knows nothing that might prevent Mittens from being a cat. Negation by failure requires that we make what is called the *closed world assumption,* which states that all relevant, true assertions are contained in our knowledge base or are derivable from assertions that are so contained. Any assertion that is not present can therefore be assumed to be false. This assumption, while often justified, can cause serious problems when knowledge bases are incomplete. We discuss this issue further in Chapter 7.

There is much to say on the topic of PROLOG-style versus LISP-style programming. A great advantage of logic programming is that the programmer need only specify rules and facts since a search engine is built directly into the language. The disadvantage is that the search control is fixed. Although it is possible to write PROLOG code that uses search strategies other than depth-first with backtracking, it is difficult to do so. It is even more difficult to apply domain knowledge to constrain a search. PROLOG does allow for rudimentary control of search through a non-logical operator called *cut.* A cut can be inserted into a rule to specify a point that may not be backtracked over.

More generally, the fact that PROLOG programs must be composed of a restricted set of logical operators can be viewed as a limitation of the expressiveness of the language. But the other side of the coin is that it is possible to build PROLOG compilers that produce very efficient code.

In the rest of this chapter, we retain the rule-based nature of PROLOG, but we relax a number of PROLOG'S design constraints, leading to more flexible rule-based architectures. Programming in PROLOG has been explained in more detail later in Chapter 25.

6.3 FORWARD VERSUS BACKWARD REASONING

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. While PROLOG only searches from a goal state, there are actually two directions in which such a search could proceed:

- Forward, from the start states
- Backward, from the goal states

The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes. Consider the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Fig. 6.2. Using those rules we could attempt to solve the puzzle shown back in Fig. 2.12 in one of two ways:

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile $n \rightarrow$
  Square 2 empty and Square 1 contains tile n
Square 1 empty and Square 4 contains tile $n \rightarrow$
  Square 4 empty and Square 1 contains tile n
Square 2 empty and Square 1 contains tile $n \rightarrow$
  Square 1 empty and Square 2 contains tile n

Fig. 6.2 *A Sample of the Rules for Solving the 8-Puzzle*

- *Reason forward from the initial states.* Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *left* sides match the root node and using their right sides to create the new

configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

- *Reason backward from the goal states.* Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning.*

Notice that the same rules can be used both to reason forward from the initial state and to reason backward from the goal state. To reason forward, the left sides (the preconditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other.

Four factors influence, the question of whether it is better to reason forward or backward:

- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

A few examples make these issues clearer. It seems easier to drive from an unfamiliar place home than from home to an unfamiliar place. Why is this? The branching factor is roughly the same in both directions (unless one-way streets are laid out very strangely). But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily, But in order to find a route from where we are to an unfamiliar place, we pretty much have to be already at the unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from the unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, it is better to reason forward using the rules for integration to try to generate an integral-free expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to head toward the largest target; this time that means chaining forward.

These two examples have illustrated the importance of the relative number of start states to goal states in determining the optimal direction in which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in some particular domain of mathematics. Our goal state is the particular theorem to be proved. Our initial states are normally a small set of axioms. Neither of these sets is significantly bigger than the other. But consider the branching factor in each of the two directions. From a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would be much better to reason backward when trying to prove theorems. Mathematicians have long realized this [Polya, 1957], as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN [Shortliffe, 1976], a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism *x*." By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would help to determine whether organism *x* is present." (For a discussion of the explanation capabilities of MYCIN, see Chapter 20.)

Most of the search techniques described in Chapter 3 can be used to search either forward or backward. By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.[2]

We can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search*. It seems appealing if the number of nodes at each step grows exponentially with the number of steps that have been taken. Empirical results [Pohl, 1971] suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results [Pohl, 1971; de Champeaux and Sint, 1977] suggest that for informed, heuristic search it is much less likely to be *so*. Figure 6.3 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on its own, to have finished. However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

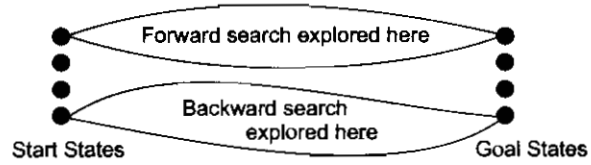


Fig. 6.3 *A Bad Use of Heuristic Bidirectional Search*

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which encodes a particular kind of knowledge.

- Forward rules, which encode knowledge about how to respond to certain input configurations.
- Backward rules, which encode knowledge about how to achieve particular goals.

[2] One exception to this is the means-ends analysis technique, described in Section 3.6, which proceeds not by making successive steps in a single direction but by reducing differences between the current and the goal states, and, as a result, sometimes reasoning backward and sometimes forward.

By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely, how it should be used in problem-solving. In the next three sections, we describe in more detail the two kinds of rule systems and how they can be combined.

6.3.1 Backward-Chaining Rule Systems

Backward-chaining rule systems, of which PROLOG is an example, are good for goal-directed problem-solving. For example, a query system would probably use backward chaining to reason about and answer user questions.

In PROLOG, rules are restricted to Horn clauses. This allows for rapid indexing because all of the rules for deducing a given fact share the same rule head. Rules are matched with the unification procedure. Unification tries to find a set of bindings for variables to equate a (sub)goal with the head of some rule. Rules in a PROLOG program are matched in the order in which they appear.

Other backward-chaining systems allow for more complex rules. In MYCIN, for example, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others. We discuss this in more detail in Chapter 8.

6.3.2 Forward-Chaining Rule Systems

Instead of being directed by goals, we sometimes want to be directed by incoming data. For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away. While this could be construed as goal-directed behavior, it is modeled more naturally by the recognize-act cycle characteristic of forward-chaining rule systems. In forward-chaining systems, left sides of rules are matched against the state description. Rules that match dump their right-hand side assertions into the state, and the process repeats.

Matching is typically more complex for forward-chaining systems than backward ones. For example, consider a rule that checks for some condition in the state description and then adds an assertion. After the rule fires, its conditions are probably still valid, so it could fire again immediately. However, we will need some mechanism to prevent repeated firings, especially if the state remains unchanged.

While simple matching and control strategies are possible, most forward-chaining systems (e.g., OPS5 [Brownston *et al.*, 1985]) implement highly efficient matchers and supply several mechanisms for preferring one rule over another. We discuss matching in more detail in the next section.

6.3.3 Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition, then forward chain on those facts to try to deduce the nature and/or cause of the disease. Now suppose that at some point, the left side of a rule was *nearly* satisfied—say, nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident. Or perhaps the tenth condition requires further medical tests. In that case, backward chaining can be used to query the user.

Whether it is possible to use the same rules for both forward-and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules, then the rules will not be reversible. Some production languages allow only reversible rules; others do not. When irreversible rules are used, then a commitment to the direction of the search must be made at the time the rules are written. But, as we suggested above, this is often a useful thing to do anyway because it allows the rule writer to add control knowledge to the rules themselves.

6.4 MATCHING

So far, we have described the process of using search to solve problems as the application of appropriate rules to individual problem states to generate new states to which the rules can then be applied, and so forth, until a solution is found. We have suggested that clever search involves choosing from among the rules that can be applied at a particular point, the ones that are most likely to lead to a solution. But we have said little about how we extract from the entire collection of rules those that can be applied at a given point. To do so requires some kind of *matching* between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule-based system. We discuss a few proposals below.

6.4.1 Indexing

One way to select applicable rules is to do a simple search through all the rules, comparing each one's preconditions to the current state and extracting all the ones that match. But there are two problems with this simple solution:

- In order to solve very interesting problems, it will be necessary to use a large number of rules. Scanning through all of them at every step of the search would be hopelessly inefficient.
- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.

Sometimes there are easy ways to deal with the first of these problems. Instead of searching through the rules, use the current state as an index into the rules and select the matching ones immediately. For example, consider the legal-move generation rule for chess shown in in Fig. 6.4. To be able to access the appropriate rules immediately, all we need do is assign an index to each board position. This can be done simply by treating the board description as a large number. Any reasonable hashing function can then be used to treat that number as an index into the rules. All the rules thai describe a given board position will be stored under the same key and so will be found together. Unfortunately, this simple indexing scheme only works because preconditions of rules match exact board configurations. Thus the matching process is easy but at the price of complete lack of generality in the statement of the rules. As discussed in Section 2.1, it is often better to write rules in a more general form, such as that shown in Fig. 6.5. When this is done, such simple indexing is not possible. In fact, there is often a trade-off between the ease of writing rules (which is increased by the use of high-level descriptions) and the simplicity of the matching process (which is decreased by such descriptions).

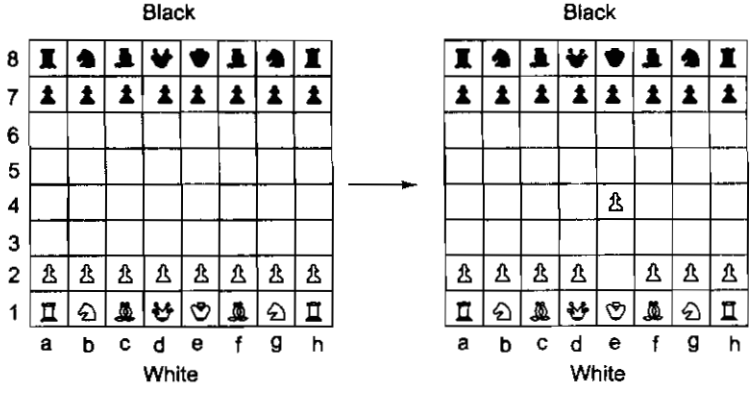


Fig. 6.4 *One Legal Chess Move*

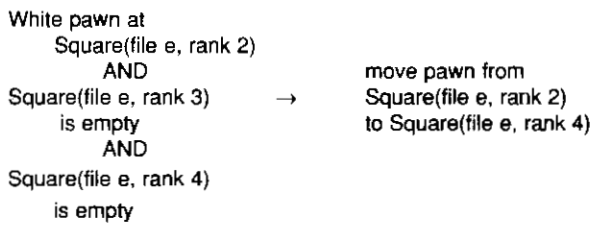


Fig. 6.5 *Another Way to Describe Chess Moves*

All of this does not mean that indexing cannot be helpful even when the preconditions of rules are stated as fairly high-level predicates. In PROLOG and many theorem-proving systems, for example, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly. In the chess example, rules can be indexed by pieces and their positions. Despite some limitations of this approach, indexing in some form is very important in the efficient operation of rule-based systems.

6.4.2 Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties (of varying complexity) that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

If we want to match a single condition against a single element in a state description, then the unification procedure of Section 5.4.4 will suffice. However, in many rule-based systems, we need to compute the whole set of rules that match the current state description. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules.[3] While it is possible to apply .unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the *many-many* match problem, in which many rules are matched against many elements in the state description simultaneously.

One efficient many-many match algorithm is RETE, which gains efficiency from three major sources:

- The temporal nature of data. Rules usually do not alter the state description radically. Instead, a rule will typically add one or two elements, or perhaps delete one or two, but most of the state description remains the same. (Recall our discussion of this as part of our treatment of the frame problem in Section 4.4.) If a rule did not match in the previous cycle, it will most likely fail to apply in the current cycle. RETE maintains a network of rule conditions, and it uses changes in the state description to determine which new rules might apply (and which rules might no longer apply). Full matching is only pursued for candidates that could be affected by incoming or outgoing data.
- Structural similarity in rules. Different rules may share a large number of pre-conditions. For example, consider rules for identifying wild animals. One rule concludes *jaguar(x)* if *mammal(x), feline(x), carnivorous(x),* and *has-spots(x).* Another rule concludes *tiger(x)* and is identical to the first rule except that it replaces *has-spots* with *has-stripes.* If we match the two rules independently, we will repeat a lot of work unnecessarily. RETE stores the rules so that they share structures in memory; sets of conditions that appear in several rules are matched (at most) once per cycle.
- Persistence of variable binding consistency. While all the individual preconditions of a rule might be met, there may be variable binding conflicts that prevent the rule from firing. For example, suppose we know the facts *son(Mary, Joe)* and *son(Bill, Bob).* The individual preconditions of the rule

[3] Conflict resolution is discussed in the next section.

$son(x, y) \wedge son(y, z) \rightarrow grandparent(x, z)$

can be matched, but not in a manner that satisfies the constraint imposed by the variable y. Fortunately, it is not necessary to compute binding consistency from scratch every time a new condition is satisfied. RETE remembers its previous calculations and is able to merge new binding information efficiently.

For more details about the RETE match algorithm, see Forgy [1982]. Other matching algorithms (e.g., Miranker [1987] and Oflazer [1987]) take different standson how much time to spend on saving state information between cycles. They can be more or less efficient than RETE, depending on the types of rules written for the domain and on the degree of hardware parallelism available.

6.4.3 Complex and Approximate Matching

A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.

An even more complex matching process is required if rules should be applied if their preconditions *approximately* match the current situation. This is often the case in situations involving physical descriptions of the world. For example, a speech- understanding program must contain rules that map from a description of a physical waveform to phones (instances of English phonemes, such as p or d). There is so much variability in the physical signal, as a result of background noise, differences in the way individuals speak, and so forth, that one can hope to find only an approximate match between the rule that describes an ideal sound and the input that describes an unideal world. Approximate matching is particularly difficult to deal with because as we increase the tolerance allowed in the match, we also increase the number of rules that will match, thus increasing the size of the main search process. But approximate matching is nevertheless superior to exact matching in situations such as speech understanding. where exact matching may often result in no rules being matched and the search process coming to a grinding halt. Although symbolic techniques for approximate matching exist, there is another, very different approach that can be used to solve this problem. We discuss it in detail in Chapter 18 where we describe connectionist systems (also called neural nets).

For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. This was the case, for example, in ELIZA [Weizenbaum, 1966], an early AI program that simulated the behavior of a Rogerian therapist. A fragment of a dialogue between ELIZA and a user is shown in Figure 6.6. ELIZA's knowledge about both English and psychology was coded in a set of simple rules. Figure 6.7 shows some ELIZA-like rules.

Person:	Men are all alike.
ELIZA:	In what way?
Person:	They're always bugging us about something or other.
ELIZA:	Can you think of a specific example?
Person:	Well, my boyfriend made me come here.
ELIZA:	Your boyfriend made you come here.
Person:	He says I'm depressed much of the time.
ELIZA:	I am sorry to hear you are depressed.
Person:	It's true. I am unhappy.
ELIZA:	Do you think coming here will help you not to be unhappy?
Person:	I need some help, that much seems certain.
ELIZA:	What would it mean to you if you got some help?
Person:	Perhaps I could learn to get along with my mother.
ELIZA:	Tell me more about your family.

Fig. 6.6 *A Bit of a Dialogue with ELIZA*

(X me Y)	→	(X you Y)
(I remember X)	→	(Why do remember X just now?)
(My {family-member} is Y)	→	(Who else in your family is Y?)
(X {family-member} Y)	→	(Tell me more about your family)

Fig. 6.7 *Some ELIZA-like rules*

ELIZA operated by matching the left sides of the rules against the user's last sentence and using the appropriate right side to generate a response. For example, if the user typed "My brother is mean to me," ELIZA might respond, "Who else in your family is mean to you?" or "Tell me more about your family." The rules were indexed by keywords so only a few had actually to be matched against a particular sentence. Some of the rules had no left side, so the rule could apply anywhere. These rules were used if no other rules matched and they generated replies such as "Tell me more about that". Notice that the rules themselves cause a form of approximate matching to occur. The patterns ask about specific words in the user's sentence. They do not need to match entire sentences. Thus a great variety of sentences can be matched by a single rule, and the grammatical complexity of English is pretty much ignored. This accounts both for ELIZA's major strength, its ability to say something fairly reasonable almost all of the time, and its major weakness, the superficiality of its understanding and its ability to be led completely astray. Approximate matching can easily lead to both these results.

As if the matching process were not already complicated enough, recall the frame problem mentioned in Chapter 4. One way of dealing with the frame problem is to avoid storing entire state descriptions at each node but instead to store only the changes from the previous node. If this is done, the matching process will have to be modified to scan backward from a node through its predecessors, looking for the required objects.

6.4.4 Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called *conflict resolution.*

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

Preferences Based on Rules

There are two common ways of assigning a preference based on the rules themselves. The first, and simplest, is to consider the rules to have been specified in a particular order, such as the physical order in which they are presented to the system. Then priority is given to the rules in the order in which they appear. This is the scheme used in PROLOG.

The other common rule-directed preference scheme is to give priority to special case rules over rules that are more general. We ran across this in Chapter 2, in the case of the water jug problem of Fig. 2.3. Recall that rules 11 and 12 were special cases of rules 9 and 5, respectively. The purpose of such specific rules is to allow for the kind of knowledge that expert problem solvers use when they solve problems directly, without search. If we consider all rules that match, then the addition of such special-purpose rules will increase the size of the search rather than decrease it. In order to prevent that, we build the matcher so that it rejects rules that are more general than other rules that also match. How can the matcher decide that one rule is more general than another? There are a few easy ways:

- If the set of preconditions of one rule contains all the preconditions of another (plus some others), then the second rule is more general than the first.
- If the preconditions of one rule are the same as those of another except that in the first case variables are specified where in the second there are constants, then the first rule is more general than the second.

Preferences Based on Objects

Another way in which the matching process can ease the burden on the search mechanism is to order the matches it finds based on the importance of the objects that are matched. There are a variety of ways this can happen. Consider again ELIZA, which matched patterns against a user's sentence in order to find a rule to generate a reply. The patterns looked for specific combinations of important keywords. Often an input sentence contained several of the keywords that ELIZA knew. If that happened, then ELIZA made use of the fact that some keywords had been marked as being more significant than others. The pattern matcher returned the match involving the highest priority keyword. For example, ELIZA knew the word "I" as a keyword. Matching the input sentence "I know everybody laughed at me" by the keyword "I" would have enabled it to respond, "You say you know everybody laughed at you." But ELIZA also knew the word "everybody" as a keyword. Because "everybody" occurs more rarely than "I," ELIZA knows it to be more semantically significant and thus to be the clue to which it should respond. So it will produce a response such as "Who in particular are you thinking of?" Notice that priority matching such as this is particularly important if only one of the choices will ever be tried. This was true for ELIZA and would also be true, say, for a person who, when leaving a fast-burning room, must choose between turning off the lights (normally a good thing to do) and grabbing the baby (a more important thing to do).

Another form of priority matching can occur as a function of the position of the matchable objects in the current state description. For example, suppose we want to model the behavior of human short-term memory (STM). Rules can be matched against the current contents of STM and then used to generate actions, such as producing output to the environment or storing something in long-term memory. In this situation, we might like to have the matcher first try to match against the objects that have most recently entered STM and only compare against older elements if the newer elements do not trigger a match. For a discussion of this method as a conflict resolution strategy in a production system, see Newell [1973].

Preferences Based on States

Suppose that there are several rules waiting to fire. One way of selecting among them is to fire all of them temporarily and to examine the results of each. Then, using a heuristic function that can evaluate each of the resulting states, compare the merits of the results, and select the preferred one. Throw away (or maybe keep for later if necessary) the remaining ones.

This approach should look familiar — it is identical to the best-first search procedure we saw in Chapter 3. Although conceptually this approach can be thought of as a conflict resolution strategy, it is usually implemented as a search control technique that operates on top of the states generated by rule applications. The drawback to this design is that LISP-coded search control knowledge is procedural and therefore difficult to modify. Many AI search programs, especially ones that learn from their experience, represent their control strategies declaratively. The next section describes some methods for capturing knowledge about control using rules.

6.5 CONTROL KNOWLEDGE

A major theme of this book is that while intelligent programs require search, search is computationally intractable unless it is constrained by knowledge about the world. In large knowledge bases that contain thousands of rules, the intractability of search is an overriding concern. When there are many possible paths of reasoning, it is critical that

```
Under conditions A and B,
Rules that do {not} mention X
        {at all,
        in their left-hand side,
        in their right-hand side}
will
        {definitely be useless,
        probably be useless
          ...
        probably be especially useful
        definitely be especially useful}
```

Fig. 6.8 *Syntax for a Control Rule [Davis, 1980]*

fruitless ones not be pursued. Knowledge about which paths are most likely to lead quickly to a goal state is often called *search control knowledge*. It can take many forms:

1. Knowledge about which states are more preferable to others.
2. Knowledge about which rule to apply in a given situation.
3. Knowledge about the order in which to pursue subgoals.
4. Knowledge about useful sequences of rules to apply.

In Chapter 3, we saw how the first type of knowledge could be represented with heuristic evaluation functions. There are many ways of representing the other types of control knowledge. For example, rules can be labeled and partitioned. A medical diagnosis system might have one set of rules for reasoning about bacteriological diseases and another set for immunological diseases. If the system is trying to prove a particular fact by backward chaining, it can probably eliminate one of the two rule sets, depending on what the fact is. Another method [Etzioni, 1989] is to assign cost and probability-of-success measures to rules. The problem-solver can then use probabilistic decision analysis to choose a cost-effective alternative at each point in the search.

By now it should be clear that we are discussing how to represent knowledge about knowledge. For this reason, search control knowledge is sometimes called *meta- knowledge*. Davis [1980] first pointed out the need for meta-knowledge, and suggested that it be represented declaratively using rules. The syntax for one type of control rule is shown in Fig. 6.8.

A number of AI systems represent their control knowledge with rules. We look briefly at two such systems, SOAR and PRODIGY.

SOAR [Laird *et al.*, 1987] is a general architecture for building intelligent systems. SOAR is based on a set of specific, cognitively motivated hypotheses about the structure of human problem solving. These hypotheses are derived from what we know about short-term memory, practice effects, etc. In SOAR:

1. Long-term memory is stored as a set of productions (or, rules).
2. Short-term memory (also called *working memory)* is a buffer that is affected by perceptions and serves as a storage area for facts deduced by rules in long-term memory. Working memory is analogous to the state description in problem solving.
3. All problem-solving activity takes place as state space traversal. There are several classes of problem-solving activities, including reasoning about which states to explore, which rules to apply in a given situation, and what effects those rules will have.
4. All intermediate and final results of problem solving are remembered (or, *chunked)* for future reference.[4]

The third feature is of most interest to us here. When SOAR is given a start state and a goal state, it sets up an initial problem space. In order to take the first step in that space, it must choose a rule from the set of applicable ones. Instead of employing a fixed conflict resolution strategy, SOAR considers that choice of

[4] We return to chunking in Chapter 17.

rules to be a substantial problem in its own right, and it actually sets up another, auxiliary problem space. The rules that apply in this space look something like the rule shown in Figure 6.8. Operator preference rules may be very general, such as the ones described in the previous section on conflict resolution, or they may contain domain-specific knowledge.

SOAR also has rules for expressing a preference for applying a whole sequence of rules in a given situation. In learning mode, SOAR can take useful sequences and build from them more complex productions that it can apply in the future.

We can also write rules based on preferences for some states over others. Such rules can be used to implement the basic search strategies we studied in Chapters 2 and 3. For example, if we always prefer to work from the state we generated last, we will get depth-first behavior. On the other hand, if we prefer states that were generated earlier in time, we will get breadth-first behavior. If we prefer any state that looks better than the current state (according to some heuristic function), we will get hill climbing. Best-first search results when state preference rules prefer the state with the highest heuristic score. Thus we see that all of the weak methods are subsumed by an architecture that reasons with explicit search control knowledge. Different methods may be employed for different problems, and specific domain knowledge can override the more general strategies.

PRODIGY [Minton *et al.*, 1989] is a general-purpose problem-solving system that incorporates several different learning mechanisms. A good deal of the learning in PRODIGY is directed at automatically constructing a set of control rules to improve search in a particular domain. We return to PRODIGY'S learning methods in Chapter 17, but we mention here a few facts that bear on the issue of search control rules. PRODIGY can acquire control rules in a number of ways:

- Through hand coding by programmers.
- Through a static analysis of the domain's operators.
- Through looking at traces of its own problem-solving behavior.

PRODIGY learns control rules from its experience, but unlike SOAR it also learns from its failures. If PRODIGY pursues an unfruitful path, it will try to come up with an explanation of why that path failed. It will then use that explanation to build control knowledge that will help it avoid fruitless search paths in the future.

One reason why a path may lead to difficulties is that subgoals can interact with one another. In the process of solving one subgoal, we may undo our solution of a previous subgoal. Search control knowledge can tell us something about the order in which we should pursue our subgoals. Suppose we are faced with the problem of building a piece of wooden furniture. The problem specifies that the wood must be sanded, sealed, and painted. Which of the three goals do we pursue first? To humans who have knowledge about this sort of thing, the answer is clear. An AI program, however, might decide to try painting first, since any physical object can be painted, regardless of whether it has been sanded. However, as the program plans further, it will realize that one of the effects of the sanding process is to remove the paint. The program will then be forced to plan a repainting step or else backtrack and try working on another subgoal first. Proper search control knowledge can prevent this wasted computational effort. Rules we might consider include:

- If a problem's subgoals include sanding and painting, then we should solve the sanding subgoal first.
- If subgoals include sealing and painting, then consider what the object is made of. If the object is made of wood, then we should seal it before painting it.

Before closing this section, we should touch on a couple of seemingly paradoxical issues concerning control rules. The first issue is called the *utility problem* [Minton, 1988]. As we add more and more control knowledge to a system, the system is able to search more judiciously. This cuts down on the number of nodes it expands. However, in deliberating about which step to take next in the search space, the system must consider all the control rules. If there are many control rules, simply matching them all can be very time-consuming. It is easy to reach a situation (especially in systems that generate control knowledge automatically)

in which the system's problem-solving efficiency, as measured in CPU cycles, is worse with the control rules than without them. Different systems handle this problem in different ways, as demonstrated in Section 17.4.4.

The second issue concerns the complexity of the production system interpreter. As this chapter has progressed, we have seen a trend toward explicitly representing more and more knowledge about how search should proceed. We have found it useful to create meta-rules that talk about when to apply other rules. Now, a production system interpreter must know how to apply various rules and meta-rules, so we should expect that our interpreters will have to become more complex as we progress away from simple backward-chaining systems like PROLOG. And yet, moving to a declarative representation for control knowledge means that previously hand coded LISP functions can be eliminated from the interpreter. In this sense, the interpreter becomes more streamlined.

SUMMARY

In this chapter, we have seen how to represent knowledge declaratively in rule-based systems and how to reason with that knowledge. We began with a simple mechanism, logic programming, and progressed to more complex production system models that can reason both forward and backward, apply sophisticated and efficient matching techniques, and represent their search control knowledge in rules.

In later chapters, we expand further on rule-based systems. In Chapter 7, we describe the use of rules that allow default reasoning to occur in the absence of specific counter evidence. In Chapter 8, we introduce the idea of attaching probabilistic measures to rules. And, in Chapter 20, we look at how rule-based systems are being used to solve complex, real-world problems.

The book *Pattern-Directed Inference Systems* [Waterman and Hayes-Roth, 1978] is a collection of papers describing the wide variety of uses to which production systems have been put in AI. Its introduction provides a good overview of the subject. Brownston *et al.* [1985] is an introduction to programming in production rules, with an emphasis on the OPS5 programming language.

EXERCISES

1. Consider the following knowledge base:
   $\forall x : \forall y : cat(x) \wedge fish(y) \rightarrow likes - to - eat(x,y)$
   $\forall x : calico(x) \rightarrow cat(x)$
   $\forall x : tuna(x) \rightarrow fish(x)$
   $tuna(Charlie)$
   $tuna(Herb)$
   $calico(Puss)$
   (a) Convert these wff's into Horn clauses.
   (b) Convert the Horn clauses into a PROLOG program.
   (c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.
   (d) Write another PROLOG program that corresponds to the same set of wff's but returns a different answer to the same query.
2. A problem-solving search can proceed either forward (from a known start state to a desired goal state) or backward (from a goal state to a start state). What factors determine the choice of direction for a particular problem?

3. If a problem-solving search program were to be written to solve each of the following types of problems, determine whether the search should proceed forward or backward:
   (a) water jug problem
   (b) blocks world
   (c) natural language understanding
4. Program the interpreter for a production system. You will need to build a table that holds the rules and a matcher that compares the current state to the left sides of the rules. You will also need to provide an appropriate control strategy to select among competing rules. Use your interpreter as the basis of a program that solves water jug problems.